

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

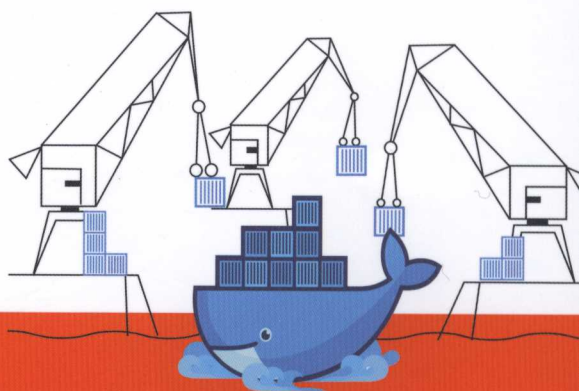
深度剖析Docker的核心概念、实现原理、应用技巧和生态系统
结合实际生产环境，通过实战案例提供有价值的应用参考

Docker

从入门到实战

Introduction and Advanced Usage of Docker

黄靖钧◎编著



- 涵盖Docker四大管理工具的基本知识，并深入分析
- 从三大组件入门应用到集群编排进阶实战，条理清晰
- 结合实际生产环境介绍上百个案例，内容都是有价值的干货
- 以Docker当前的流行版本为例讲解Swarm集群管理



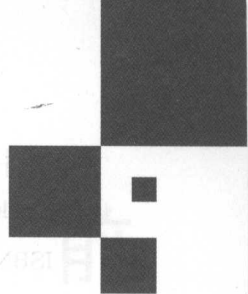
机械工业出版社
China Machine Press

内容简介

本书从Docker的相关概念与基础知识讲起，结合实际应用，通过不同开发环境的实战例子，详细介绍了Docker的基础知识与进阶实战的相关内容，以引领读者快速入门并提高。

本书共19章，分3篇。第1篇容器技术与Docker概念，涵盖的内容有容器技术、Docker简介、安装与测试Docker等。第2篇Docker基础知识，涵盖的内容有Docker基础、Docker镜像、Dockerfile文件、Docker仓库、Docker容器、数据卷、网络管理等。第3篇Docker进阶实战，涵盖的内容有操作系统、编排工具Compose、Web服务器与应用、数据库、编程语言、Docker API、私有仓库、集群网络、Docker安全等。

本书非常适合所有对Docker感兴趣的入门新手阅读。不管是开发人员还是运维人员，都可以通过本书学习Docker的基本知识。即使不是程序员的读者，本书同样适合。普通用户完全可以把Docker作为一个“好玩的工具”来使用，以体验Docker带来的便捷。

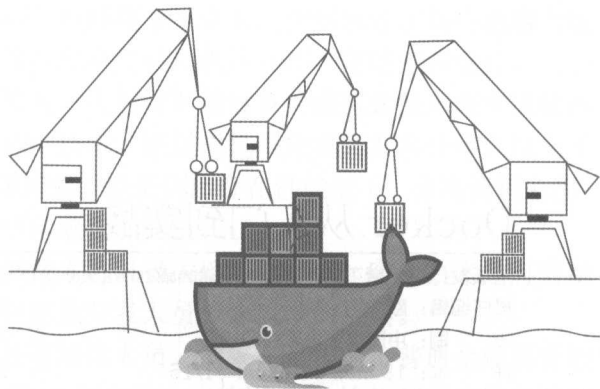


Docker

从入门到实战

Introduction and Advanced Usage of Docker

黄靖钧◎编著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Docker 从入门到实战 / 黄靖钧编著. —北京: 机械工业出版社, 2017.6

ISBN 978-7-111-57328-9

I. D… II. 黄… III. Linux 操作系统—程序设计 IV. TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 166174 号

Docker 从入门到实战

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 欧振旭

责任校对: 姚志娟

印 刷: 中国电影出版社印刷厂

版 次: 2017 年 8 月第 1 版第 1 次印刷

开 本: 185mm×260mm 1/16

印 张: 22.75

书 号: ISBN 978-7-111-57328-9

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

前言

Docker 作为一个 2013 年才诞生的开源项目，其发展的速度和火爆程度却令人惊叹。容器技术本不是什么新鲜事物，但是在 Docker 的整合下，一切变得清晰、易用起来，并且随着各大云计算厂商的进场，使 Docker 得到了极大的推广。

如今，Docker 已经成为容器技术领域当仁不让的领头羊。国内外以 Docker 技术起家的创业公司如雨后春笋般涌现出来，体现了容器市场的巨大需求。越来越多的企业开始逐步把传统的应用开发流程迁移到 Docker 容器中作为开发部署流程的一环。伴随而来的是各种复杂的需求与 Docker 尚不算完善的功能所产生的矛盾，这些问题制约着企业容器化的脚步。

另一方面，Docker 以其友好的使用体验使广大开发者对其“一见倾心”，越来越多的开发者使用 Docker 作为应用分发部署的一个重要阵地。尽管如此，Docker 对于大部分开发者而言还是尚未开拓的疆土。特别是对于国内环境而言，Docker 的推广基本上靠国内几家与 Docker 相关的初创公司。本书以一位普通的全栈开发者的身份，详细介绍了 Docker 的基础知识，分享了企业级容器云的实战经验。

为什么学习 Docker

如果您是一名开发者，想必遇到过“这个程序只有在我的机器上才可以运行”的情况。随着用户需求变得多样，软件愈发复杂，所依赖环境愈发庞大，使得软件在其他机器上运行需要做大量的迁移工作。更糟糕的是，这些琐事完成后软件还不一定能正常运行。

为了解决这些问题，虚拟化技术开始普及。人们可以通过各种虚拟化技术来实现软件的迁移和分发。最常见的就是虚拟机或 KVM 技术。在虚拟机里完成开发再迁移到线上不会出现环境问题，解决了迁移过程中的诸多难题，但是仍然存在性能低下、分发流程麻烦、耗时和成本昂贵等问题。在云计算时代这些问题更加突出。

随着容器技术的普及，人们意识到容器技术可以极大地降低成本。容器技术具有启动快、体积小和分发迅速等诸多特点，这简直就是开发人员梦寐以求的工具。

而“欣喜若狂”的不止是开发人员，还有运维人员。如果在十年前，普通企业要管理上百台服务器，最可能使用的方法是通过 Shell 脚本的方式使用 SSH 连接到所有服务器然后执行相同的指令，并把日志保存起来归档。这种方式我们称之为第一代运维。那时维护服务器是一项繁重的工作，工程师不得不把大量的时间耗费在服务器管理上。

随着技术的发展，虚拟化技术的普及和云计算的出现，企业需要管理的服务器数量大幅增长。过去我们只要管理企业内部数据中心的物理服务器，而现在则要管理遍布全球的服务器，运维成本愈发昂贵。于是开发者开始针对云计算时代服务器运维方式做出改变，涌现出了诸如 Ansible、Puppet、SaltStack 和 Chef 等出色的运维工具。人们可以通过这些

工具快速地完成对上百台甚至上千台服务器的管理操作。这被称之为第二代运维。它极大地解决了管理庞大服务器集群的难题，使人们可以在屏幕面前通过一个界面管理所有服务器。但本质上这些工具都是通过 SSH 或者类似于 SSH 的方式连接到服务器来管理服务器集群，这意味着其实第二代运维和第一代运维并没有发生根本性的改变。

上面那些运维工具在云计算普及的大势下很快暴露了它们的问题——速度。因为大部分运维工具依靠的是 SSH 连接来交换信息，这使得整个过程十分耗时，更不用说其他复杂的管理操作。而随着容器技术的爆发，以 Docker 为代表的容器技术开始发力，并随着 DevOps 概念的普及，使运维发生了根本性的改变。容器集群管理不再是通过低效的 SSH 来连接服务器，甚至不需要登录服务器就可以完成对服务器的管理。人们发现，通过容器管理集群可以抛弃传统的“SSH+秘钥”的连接方式来连接服务器，这对大规模集群来说是一个极大的变革。而且在速度上，容器技术在上百台服务器上启动应用只需要一眨眼的时间，这使得运维的工作大大减轻。

运维和开发在容器时代逐步“融为一体”，形成一个流水线车间的工作环境。这对于软件行业来说无疑是一次巨大的变革。

如果您也对传统的软件开发和运维的烦琐流程感到吃力，又对容器技术感兴趣，那么本书将是很好的入门书籍。

如果您不是职业的开发和运维人员，对 Linux 也不算熟悉，但属于一个对 Docker 感兴趣的极客，想通过 Docker 部署一些复杂的应用，本书也一样适合您。本书虽以 Linux 为平台介绍 Docker 的使用，但是与在 Windows 和 Mac OS 平台上的操作基本一致，普通用户完全可以把 Docker 当做一个“好玩的工具”来使用，体验 Docker 带来的便捷。

本书特色

- **适合新手入门。**本书在基础方面内容非常详尽，包括镜像的构建、容器的运行监控、网络的管理、仓库的应用、集群的部署等内容，全面、细致地介绍了 Docker 的基本使用方法与实现原理，适合新手入门。
- **应用结合实际。**本书在实战应用部分结合实际情况，从不同的角度分析问题并提出对应的解决办法，扩展了很多实用的实战技巧。实战部分根据不同类型的开发环境构建基础开发环境镜像，使读者可以直接使用 Docker 进入测试开发，并根据不同类型的部署做了详细介绍。
- **范例丰富。**在实战章节中的范例皆由浅入深，全面、实用且不缺乏趣味性，有助于读者了解其内部原理，进而应用到其他项目的思考与开发中。全书的代码均有指明出处以及链接，读者可以通过文中链接找到源代码。
- **版本最新。**本书使用目前流行的 Docker 1.12 版本，紧跟 Docker 更新步伐，介绍了新的 Docker Swarm 集群管理方式。

本书内容体系

第1篇 容器技术与Docker概念（第1~3章）

本篇主要介绍了容器技术的发展历史与容器技术的原理，并解释了 Docker 与其他容器

技术的区别,对比了 Docker 与虚拟机的异同,客观地评价了两者的优缺点。另外,本篇还介绍了 Docker 分别在 Linux、Windows 和 Mac OS 系统下的安装方法,以及二进制安装方法。

第2篇 Docker基础知识(第4~10章)

本篇主要介绍了 Docker 的基本操作及简单应用,包括 Docker 基础、Docker 镜像、Docker file 文件、Docker 仓库、Docker 容器、数据卷的使用方法与原理及网络管理等内容。通过对本篇内容的学习,读者可以掌握最常用的 Docker 知识。

第3篇 Docker进阶实战(第11~19章)

本篇包含了许多 Docker 在实际开发中的应用实例,包括操作系统、编排工具 Compose、Web 服务器与应用、数据库、编程语言、Docker API、私有仓库、集群网络、Docker 安全等内容,详细讲解了 Docker 在容器云环境中的应用。读者通过这部分内容的学习已经完全可以在实际生产环境中应用 Docker 了。

本书读者对象

- Dock 开发入门人员;
- 容器技术爱好者;
- 各类运维人员;
- 基于 Docker 构建云计算平台的技术人员;
- 大、中专院校的学生;
- 相关培训学校的学员。

本书配套资源及获取方式

本书涉及的源代码文件等配套学习资源需要读者自行下载。请读者登录机械工业出版社华章公司的网站 www.hzbook.com, 然后搜索到本书页面, 按照页面上的说明进行下载即可。

本书作者

本书由黄靖钧主笔编写。其他参与编写的人员有张昆、张友、赵桂芹、晁楠、高彩琴、郭现杰、刘琳、王凯迪、王晓燕、吴金艳、尹继平、张宏霞、张晶晶、陈冠军、魏春、张燕、范陈琼、孟春燕、王晓玲、项宇峰、肖磊鑫、薛楠、杨丽娜、闫利娜、王韶、李杨坡、刘春华、黄艳娇、刘雁。

本书的顺利出版,要感谢机械工业出版社华章公司各位编辑的辛勤劳动和付出,另外对网络上提供有益资料的众多作者也在此表示感谢。

虽然我们对本书中所述内容都尽量核实,并多次进行文字校对,但因时间所限,加之水平所限,书中疏漏和错误在所难免,敬请广大读者批评指正。

目录

前言

第 1 篇 容器技术与 Docker 概念

第 1 章 容器技术	2
1.1 什么是容器	2
1.1.1 关于虚拟化	2
1.1.2 容器的定义	3
1.1.3 为什么使用容器	3
1.2 容器技术的前世今生	4
1.2.1 容器技术的起源	4
1.2.2 容器技术的发展	5
1.3 容器的原理	7
1.3.1 从 namespace 说起	7
1.3.2 认识 Cgroups	9
1.3.3 容器的创建	11
1.4 容器云	12
1.5 容器与 Docker	13
1.6 本章小结	13
第 2 章 Docker 简介	14
2.1 什么是 Docker	14
2.1.1 Docker 的历史	14
2.1.2 Docker 的现状	16
2.1.3 Docker 的未来	17
2.2 Docker 的功能及优缺点	18
2.2.1 Docker 在解决什么	18
2.2.2 为什么选择 Docker	19
2.2.3 Docker 的缺点	19
2.3 Docker 和虚拟机	19
2.3.1 Docker 与虚拟机的区别	20
2.3.2 Docker 与虚拟机的优缺点	20
2.4 Docker 与 runC	21
2.4.1 libcontainer 与 runC	21
2.4.2 runC 的使用	22

2.4.3	runC 原理	22
2.5	Docker 基本架构	24
2.5.1	Docker Client 介绍	24
2.5.2	Docker daemon 介绍	25
2.5.3	Docker 镜像	25
2.5.4	Docker 容器	26
2.5.5	Docker 仓库	26
2.6	本章小结	26
第 3 章	安装 Docker	27
3.1	Linux 系统	27
3.1.1	一键安装脚本	27
3.1.2	Debian 发行版	28
3.1.3	Ubuntu 发行版	30
3.1.4	Centos/Fedora 发行版	33
3.1.5	Arch Linux 发行版	37
3.1.6	Suse/openSUSE 发行版	38
3.2	Windows 与 Mac OS 系统	38
3.2.1	在 Windows 上安装原生 Docker	39
3.2.2	在 Mac OS 上安装原生 Docker	41
3.3	二进制安装	43
3.3.1	获取 Linux 二进制包	44
3.3.2	获取 Mac OS X 二进制包	44
3.3.3	获取 Windows 的二进制包	45
3.3.4	树莓派安装 Docker	45
3.4	本章小结	46

第 2 篇 Docker 基础知识

第 4 章	Docker 基础	48
4.1	Docker 基本操作	48
4.1.1	依附容器的 docker attach 命令	49
4.1.2	构建镜像的 docker build 命令	51
4.1.3	提交容器的 docker commit 命令	52
4.1.4	复制文件到宿主机的 docker cp 命令	52
4.1.5	创建容器的 docker create 命令	53
4.1.6	查看容器变化的 docker diff 命令	54
4.1.7	查看事件的 docker events 命令	54
4.1.8	进入容器的 docker exec 命令	55
4.1.9	导出容器的 docker export 命令	56
4.1.10	查看镜像历史的 docker history 命令	56
4.1.11	查看本地镜像的 docker images 命令	57
4.1.12	导入容器的 docker import 命令	58

4.1.13	查看 Docker 信息的 docker info 命令	58
4.1.14	查看各项详细信息的 docker inspect 命令	59
4.1.15	杀死容器的 docker kill 命令	60
4.1.16	导入镜像的 docker load 命令	60
4.1.17	登录仓库的 docker login 命令	61
4.1.18	登出仓库的 docker logout 命令	61
4.1.19	查看容器日志的 docker logs 命令	62
4.1.20	管理网络的 docker network 命令	62
4.1.21	管理节点的 docker node 命令	63
4.1.22	暂停容器的 docker pause 命令	64
4.1.23	查看容器端口的 docker port 命令	64
4.1.24	查看本地容器信息的 docker ps 命令	65
4.1.25	拉取镜像的 docker pull 命令	65
4.1.26	推送镜像的 docker push 命令	66
4.1.27	重命名容器的 docker rename 命令	66
4.1.28	重启容器的 docker restart 命令	66
4.1.29	删除容器的 docker rm 命令	67
4.1.30	删除镜像的 docker rmi 命令	67
4.1.31	运行容器的 docker run 命令	68
4.1.32	导出镜像的 docker save 命令	72
4.1.33	搜索镜像的 docker search 命令	73
4.1.34	管理服务的 docker service 命令	74
4.1.35	启动容器的 docker start 命令	74
4.1.36	查看容器状态的 docker stats 命令	75
4.1.37	停止容器的 docker stop 命令	75
4.1.38	管理集群的 docker swarm 命令	76
4.1.39	设置镜像标签的 docker tag 命令	76
4.1.40	查看容器进程的 docker top 命令	77
4.1.41	恢复暂停容器的 docker unpause 命令	77
4.1.42	更新容器的 docker update 命令	77
4.1.43	查看 Docker 版本的 docker version 命令	78
4.1.44	管理数据卷的 docker volume 命令	78
4.1.45	设置等待的 docker wait 命令	79
4.2	启动第一个 Docker 容器	79
4.3	构建第一个 Docker 镜像	80
4.4	本章小结	81
第 5 章	Docker 镜像	82
5.1	认识镜像	82
5.1.1	使用 docker pull 拉取镜像	82
5.1.2	搜索镜像	83
5.1.3	查看镜像信息	84
5.2	创建镜像	86
5.2.1	剖析 Hello World 镜像	86
5.2.2	从 Dockerfile 构建镜像	86

5.2.3	自动构建镜像	87
5.2.4	提交容器为镜像	90
5.3	导出和导入镜像	91
5.3.1	导出镜像到本地文件系统	91
5.3.2	从本地文件系统导入镜像	91
5.4	发布镜像	91
5.4.1	发布镜像到 Docker Hub	92
5.4.2	给镜像打上标签	92
5.4.3	发布到第三方镜像仓库	92
5.5	删除镜像	93
5.5.1	删除本地镜像	93
5.5.2	删除仓库镜像	93
5.6	Docker 镜像扩展	94
5.6.1	Docker 镜像里有什么	94
5.6.2	Docker 镜像的存储方式	95
5.6.3	联合挂载	95
5.6.4	Git 式管理	96
5.7	本章小结	96
第 6 章	Dockerfile 文件	97
6.1	Dockerfile 基本结构	97
6.1.1	Dockerfile 基础	97
6.1.2	Dockerfile 的书写规则	98
6.1.3	基础镜像信息和维护者信息	99
6.2	Dockerfile 指令	99
6.2.1	指定基础镜像的 FROM 指令	99
6.2.2	设置维护者信息的 MAINTAINER 指令	99
6.2.3	执行构建命令的 RUN 指令	99
6.2.4	设置镜像环境变量的 ENV 指令	100
6.2.5	复制文件的 COPY 指令	100
6.2.6	添加文件的 ADD 指令	100
6.2.7	指定端口暴露的 EXPOSE 指令	100
6.2.8	设置镜像启动命令的 CMD 指令	101
6.2.9	设置接入点的 ENTRYPOINT 指令	102
6.2.10	设置数据卷的 VOLUME 指令	102
6.2.11	设置构建用户的 USER 指令	103
6.2.12	设置工作目录的 WORKDIR 指令	103
6.2.13	设置二次构建指令的 ONBUILD 指令	104
6.2.14	设置元数据的 LABEL 指令	105
6.2.15	设置构建变量的 ARG 指令	105
6.2.16	设置停止信号的 STOPSIGNAL 指令	105
6.2.17	检查镜像状态的 HEALTHCHECK 指令	105
6.2.18	设置命令执行环境的 SHELL 指令	106
6.3	镜像构建实战	106
6.3.1	收集应用信息	106

6.3.2 编写 Dockerfile	106
6.3.3 设置自动构建	107
6.4 本章小结	108
第 7 章 Docker 仓库	109
7.1 官方仓库 Docker Hub	109
7.1.1 Docker Hub 登录与使用	109
7.1.2 Docker Hub 与 Docker Cloud	110
7.2 国内镜像仓库	111
7.2.1 中国科学技术大学镜像仓库	111
7.2.2 DaoCloud 镜像加速器	112
7.2.3 阿里云镜像加速器	113
7.2.4 灵雀云镜像加速器	114
7.2.5 时速云镜像加速器	115
7.2.6 网易蜂巢	116
7.2.7 自建镜像加速器	116
7.3 私有仓库	117
7.3.1 搭建私有仓库	117
7.3.2 私有仓库的使用	117
7.3.3 私有仓库安全性	118
7.4 Registry 原理	118
7.4.1 Registry 组成	118
7.4.2 Registry 工作流程	119
7.5 本章小结	120
第 8 章 Docker 容器	121
8.1 容器基本操作	121
8.1.1 创建容器	122
8.1.2 启动容器	122
8.1.3 后台运行容器	123
8.1.4 自动重启容器	123
8.1.5 停止与杀死容器	124
8.1.6 删除容器	125
8.1.7 查看容器信息	125
8.2 进入容器内部	128
8.2.1 使用 attach 进入容器	128
8.2.2 使用 exec 进入容器	129
8.2.3 使用 nsenter 进入容器	129
8.3 导出和导入容器	130
8.3.1 导出容器	130
8.3.2 导入容器	130
8.4 容器结构	131
8.4.1 容器格式是什么	131
8.4.2 容器内部结构	132
8.5 本章小结	133

第 9 章 数据卷	134
9.1 数据卷是什么	134
9.1.1 数据卷介绍	134
9.1.2 数据卷容器介绍	135
9.2 为容器挂载数据卷	135
9.2.1 挂载数据卷	135
9.2.2 挂载数据卷容器	136
9.2.3 数据卷挂载小结	137
9.3 备份、恢复、迁移数据卷	139
9.3.1 备份数据卷	139
9.3.2 迁移、恢复数据卷	140
9.4 容器数据卷扩展	140
9.4.1 卷插件介绍	140
9.4.2 Convoy 的使用	140
9.4.3 Flocker 的使用	141
9.5 本章小结	142
第 10 章 网络管理	143
10.1 Docker 网络基础	143
10.1.1 端口映射	143
10.1.2 端口暴露	145
10.1.3 容器互联	146
10.2 Docker 网络模式	147
10.2.1 none 模式	147
10.2.2 container 模式	149
10.2.3 host 模式	150
10.2.4 bridge 模式	151
10.2.5 overlay 模式	152
10.3 Docker 网络配置	152
10.3.1 Daemon 网络参数	152
10.3.2 配置 DNS	153
10.3.3 network 命令	154
10.4 本章小结	154

第 3 篇 Docker 进阶实战

第 11 章 操作系统	156
11.1 Alpine 发行版	156
11.1.1 官方镜像	156
11.1.2 运行 Alpine Linux	157
11.1.3 构建基于 Alpine Linux 的镜像	157
11.1.4 Alpine Linux 软件仓库	158
11.2 Busybox 发行版	159

11.2.1	官方镜像	159
11.2.2	运行 Busybox	159
11.2.3	构建基于 Busybox 的镜像	159
11.3	Debian/Ubuntu 发行版	160
11.3.1	官方镜像	160
11.3.2	运行 Debian/Ubuntu	160
11.3.3	构建基于 Debian/Ubuntu 的镜像	161
11.4	CentOS/Fedora 发行版	162
11.4.1	官方镜像	162
11.4.2	运行 CentOS/Fedora	162
11.5	CoreOS 发行版	163
11.5.1	为什么使用 CoreOS	163
11.5.2	用 Vagrant 安装 CoreOS	163
11.6	RancherOS 发行版	165
11.6.1	为什么使用 RancherOS	165
11.6.2	在服务器安装 RancherOS	166
11.6.3	基于 RancherOS 的 Docker 管理	167
11.7	本章小结	167
第 12 章	编排工具 Compose	169
12.1	安装 Docker Compose	169
12.1.1	二进制安装	169
12.1.2	使用 Python pip 安装	169
12.2	Compose 命令基础	170
12.2.1	指定配置文件	171
12.2.2	指定项目名称	171
12.2.3	Compose 环境变量	171
12.2.4	构建服务镜像的 build 命令	172
12.2.5	生成 DAB 包的 bundle 命令	173
12.2.6	检查配置语法的 config 命令	173
12.2.7	创建服务容器的 create 命令	174
12.2.8	清理项目的 down 命令	174
12.2.9	查看事件的 events 命令	175
12.2.10	进入服务的 exec 命令	176
12.2.11	杀死服务容器的 kill 命令	176
12.2.12	查看服务容器日志的 logs 命令	176
12.2.13	暂停服务容器的 pause 命令	177
12.2.14	查看服务容器端口状态的 port 命令	177
12.2.15	查看项目容器信息 ps 命令	177
12.2.16	拉取项目镜像的 pull 命令	178
12.2.17	推送项目镜像的 push 命令	179
12.2.18	重启服务容器的 restart 命令	179
12.2.19	删除项目容器的 rm 命令	179
12.2.20	执行一次性命令的 run 命令	180
12.2.21	设置服务容器数量的 scale 命令	182

12.2.22	启动服务容器的 start 命令	184
12.2.23	停止服务容器的 stop 命令	184
12.2.24	取消暂停的 unpause 命令	185
12.2.25	启动项目的 up 命令	185
12.3	Compose 配置文件	186
12.3.1	配置文件基础	186
12.3.2	基本配置	187
12.3.3	网络配置	196
12.3.4	配置扩展	197
12.4	Compose 实战	200
12.4.1	部署 Django	200
12.4.2	部署 Rails	202
12.4.3	部署 WordPress	203
12.5	本章小结	205
第 13 章	Web 服务器与应用	206
13.1	Apache 服务器	206
13.1.1	官方镜像	206
13.1.2	运行官方镜像	208
13.1.3	基于 Ubuntu 构建 Apache 镜像	209
13.1.4	基于 Alpine 构建 Apache 镜像	210
13.1.5	第三方优质镜像	211
13.2	Nginx 服务器	212
13.2.1	官方镜像介绍	212
13.2.2	运行官方镜像	212
13.2.3	构建 Nginx 镜像	214
13.2.4	第三方镜像推荐	216
13.3	Tomcat 服务器	216
13.3.1	官方镜像介绍	217
13.3.2	运行官方镜像	217
13.3.3	构建 Tomcat 镜像	218
13.4	其他 Web 服务器	220
13.4.1	Caddy 服务器	220
13.4.2	WebLogic 服务器	221
13.5	本章小结	223
第 14 章	数据库	224
14.1	MySQL 数据库	224
14.1.1	官方镜像的剖析与使用	224
14.1.2	构建自己的 MySQL 镜像	226
14.2	PostgreSQL 数据库	228
14.2.1	官方镜像的使用	228
14.2.2	官方镜像的剖析	230
14.2.3	构建 PostgreSQL 镜像	232
14.2.4	数据备份与导入	234
14.3	Redis 数据库	235

14.3.1 官方镜像的使用	235
14.3.2 构建自己的 Redis 镜像	236
14.4 MongoDB 数据库	238
14.4.1 官方镜像的使用	238
14.4.2 构建自己的 MongoDB 镜像	239
14.5 其他	240
14.5.1 在容器中使用 SQLite	240
14.5.2 构建自己的 MariaDB 镜像	241
14.5.3 使用 Docker 部署 Oracle XE 数据库	243
14.6 本章小结	244
第 15 章 编程语言	245
15.1 C/C++ 语言	245
15.1.1 官方镜像 library/gcc	245
15.1.2 基于 Alpine 构建 C/C++ 镜像	246
15.2 Golang 语言	247
15.2.1 官方镜像 library/golang	248
15.2.2 Beego 框架	250
15.2.3 自助 Git 服务——Gogs	250
15.2.4 基于 Alpine 构建 Golang 镜像	252
15.3 Java 语言	253
15.3.1 官方镜像 library/openjdk	253
15.3.2 基于 Alpine 构建 Java 镜像	254
15.3.3 Tomcat 服务器	255
15.3.4 下一代集成开发环境——Eclipse Che	256
15.4 JavaScript (Node.js) 语言	258
15.4.1 官方镜像 library/node	258
15.4.2 vue.js 框架	260
15.4.3 Express 框架	261
15.4.4 浏览器里的 IDE——Cloud9-IDE	262
15.4.5 基于 Alpine 构建 Node.js 镜像	264
15.5 PHP 语言	265
15.5.1 官方镜像 library/php	265
15.5.2 快速安装扩展	267
15.5.3 LNMP 环境组合	268
15.5.4 基于 Alpine 构建 PHP 镜像	270
15.5.5 自建私有云存储——ownCloud	271
15.5.6 Typecho 博客系统	273
15.6 Python 语言	275
15.6.1 官方镜像 library/python	275
15.6.2 Flask 框架	275
15.6.3 基于 Alpine 构建 Python 镜像	277
15.7 Swift 语言	277
15.7.1 构建 Swift 镜像	277
15.7.2 Kitura 框架	278

15.8 本章小结	280
第 16 章 Docker API 介绍	281
16.1 认识 Docker API	281
16.1.1 RESTful 介绍	281
16.1.2 开启 socket	282
16.1.3 使用 curl	283
16.1.4 使用 Postman	284
16.2 Docker Remote API 介绍	286
16.2.1 容器 API	286
16.2.2 镜像 API	293
16.3 其他 API	299
16.3.1 常用 API	300
16.3.2 Trusted Registry API 介绍	302
16.4 本章小结	304
第 17 章 私有仓库	305
17.1 Docker Registry 介绍	305
17.1.1 部署 Docker Registry	305
17.1.2 私有仓库的 push 与 pull	305
17.1.3 配置 Registry	306
17.1.4 添加 Docker Hub Mirror 功能	311
17.2 认证与前端	312
17.2.1 设置反代理	312
17.2.2 为私有仓库添加认证服务	313
17.2.3 为私有仓库添加可视化界面	315
17.3 企业级私有仓库 Harbor	316
17.3.1 Harbor 配置详解	316
17.3.2 配置 HTTPS	318
17.3.3 使用 Compose 启动 Harbor	320
17.4 私有仓库前端授权工具 Portus	321
17.4.1 一键部署 Portus	321
17.4.2 手动配置	321
17.4.3 启动 Portus	322
17.5 本章小结	323
第 18 章 集群网络	324
18.1 Swarm 集群	324
18.1.1 认识 Swarm	324
18.1.2 建立跨主机网络	328
18.1.3 在跨主机网络上部署应用	328
18.1.4 Swarm 集群负载	331
18.2 第三方网络管理工具	333
18.2.1 Weave 介绍	333
18.2.2 Flannel 介绍	335
18.2.3 Pipwork 介绍	335

18.3 第三方服务发现.....	336
18.3.1 Etcd 介绍	336
18.3.2 Consul 介绍	337
18.4 第三方集群管理.....	337
18.4.1 Kubernetes 介绍	337
18.4.2 Mesos Shipyard 介绍.....	339
18.5 本章小结.....	339
第 19 章 Docker 安全	340
19.1 Docker 安全机制.....	340
19.1.1 Daemon 安全	340
19.1.2 容器与镜像安全.....	341
19.2 Docker 资源控制.....	342
19.2.1 限制 CPU.....	342
19.2.2 限制内存.....	343
19.2.3 限制 I/O	343
19.2.4 文件系统防护.....	344
19.2.5 镜像瘦身神器 Docker Slim	344
19.2.6 强制访问控制工具 SELinux 或 AppArmor.....	344
19.3 本章小结.....	345

第 1 篇

容器技术与 Docker 概念

» 第 1 章 容器技术

» 第 2 章 Docker 简介

» 第 3 章 安装 Docker

在云计算和虚拟化技术快速发展的今天，容器技术作为一种轻量级的虚拟化技术，正逐渐成为云计算和虚拟化技术的重要组成部分。容器技术通过将操作系统与应用程序分离，使得应用程序可以在不同的操作系统和硬件平台上运行，从而实现了应用程序的跨平台运行和部署。容器技术的出现，极大地简化了应用程序的部署和管理，提高了应用程序的灵活性和可扩展性。在云计算和虚拟化技术中，容器技术扮演着越来越重要的角色。本文将介绍容器技术的基本概念、容器技术的优缺点、容器技术的应用场景以及容器技术的未来发展趋势。通过本文的学习，读者可以了解容器技术的基本原理和应用，为后续的学习和实践打下坚实的基础。

容器技术可以理解为一种轻量级的虚拟化技术，它允许在同一个操作系统内核上运行多个隔离的应用程序。容器技术通过将应用程序与操作系统分离，使得应用程序可以在不同的操作系统和硬件平台上运行，从而实现了应用程序的跨平台运行和部署。

1.1.3 为什么使用容器

容器技术的出现，极大地简化了应用程序的部署和管理，提高了应用程序的灵活性和可扩展性。在云计算和虚拟化技术中，容器技术扮演着越来越重要的角色。本文将介绍容器技术的基本概念、容器技术的优缺点、容器技术的应用场景以及容器技术的未来发展趋势。通过本文的学习，读者可以了解容器技术的基本原理和应用，为后续的学习和实践打下坚实的基础。

第 1 章 容器技术

随着容器技术的长足发展，特别是 Docker 的流行，容器技术已经被越来越多的企业应用于生产环境中。在了解 Docker 之前，需要先了解一下容器技术。

本章将详细介绍容器技术，逐步认识容器的原理，并尝试启动简单的容器。在学习了解容器技术的发展历程。

本章主要包括 3 部分：

- 认识容器技术，了解容器技术的发展历程。
- 了解容器技术的概念及基本原理。
- 了解容器与容器云对软件行业的影响。

1.1 什么是容器

容器技术并不是一个全新的概念，它又称为容器虚拟化。显然它是虚拟化技术中的一种。虚拟化技术目前主要有硬件虚拟化、半虚拟化和操作系统虚拟化等。本书讲述的容器虚拟化属于操作系统虚拟化，其相较于其他主流虚拟化技术更轻量。

1.1.1 关于虚拟化

虚拟化技术的分类与定义在不同领域有不同理解。对于计算机领域，虚拟化技术主要分为两大类：一类基于硬件虚拟化，另一类基于软件虚拟化。硬件虚拟化并不多见，大都是半虚拟化与软件结合，应用较为广泛的则是基于软件的虚拟化技术。

基于软件虚拟化又可以分为应用虚拟化（如 Wine）和平台虚拟化（如虚拟机）。本书中的容器技术属于操作系统虚拟化，属于平台虚拟化的一种。

从图 1.1 中可以看到，Docker 属于容器技术的一种，而容器技术属于操作系统虚拟化的一种，有时这种分类会因为技术的发展而有变动。

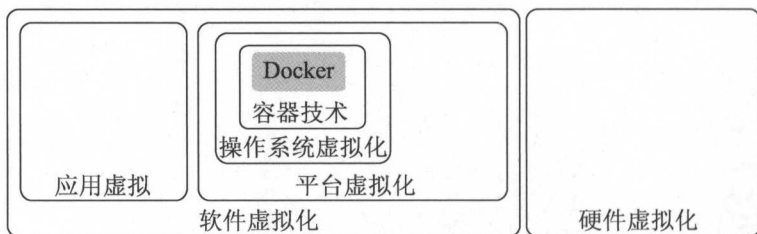


图 1.1 容器技术在虚拟化技术的位置

1.1.2 容器的定义

所谓容器，顾名思义就是用来放东西的道具。有意思的是，在 Docker 刚进入国内时，还有过一段时间在讨论 Container 这个单词是翻译为“容器”合适，还是翻译为“集装箱”合适。

之所以有人建议翻译为“集装箱”，并不仅仅是因为 Docker 的图标是一条鲸鱼驮着几个集装箱的形象（如图 1.2 所示），还因为容器技术本身就是借鉴了工业运输的经验发展而来。



图 1.2 容器技术与集装箱

《经济学家》这样评价工业运输领域的集装箱：“没有集装箱，就不可能有全球化。”在 1956 年集装箱出现之前，货物运输缺乏标准，成本很高。特别是远洋运输。直到“集装箱”这个概念的出现，毫不起眼的集装箱降低了货物运输的成本，实现了货物运输的标准化，并以此为基础逐步建立全球范围内的船舶、港口、航线、公路、中转站、桥梁、隧道、多式联运相配套的物流系统，世界经济形态因此而改变。

同样，软件行业的容器技术也是在尝试打造一套标准化的软件构建、分发流程，以降低运维成本，提高软件安全与运行稳定等。与工业运输的集装箱不同，容器技术要复杂得多。它不仅仅是要打造一个运输用的“集装箱”，还要保证软件在容器内能够运行，在操作系统上打造一个“独立的箱子”。这需要解决文件系统、网络、硬件等多方面的问题。经过长时间的发展，容器技术已经逐步成熟，并在 Docker 的诞生下迎来它的繁荣时代。

读者大可把容器理解为一个沙盒，每个容器是独立的，容器之间可以相互通信。

1.1.3 为什么使用容器

与传统软件行业的开发和运维相比，容器虚拟化可以更高效地构建应用，也更容易管理维护。举个简单的例子，常见的 LAMP 组合开发网站，按照传统的做法自然是各种安装，然后配置、测试、发布，中间麻烦事一大堆，相信不少读者深有体会。

当服务器需要搬迁时，往往需要再执行一次以前的部署步骤，极大地浪费了运维人员的时间。最可怕的是搬迁后往往因为一些不可预知的原因而导致软件无法正常运行，只能一头扎进代码中找 Bug。

如果使用了容器技术，运维只需要一句简单的命令即可部署一整套 LAMP 环境，并且不需要复杂的配置与测试。即便搬迁也只是打包传输即可。即使在另一台机器上，软件也不会出现“水土不服”的情况。这无疑节约了运维人员的大量时间。

而对于开发者来说，一处构建，到处运行，大概是梦寐以求的事情。这也是很多跨平台语言的宣传标语之一。但不管是怎样的跨平台语言，在很多细节上都需要不少调整才能运行在另一个平台上。但容器技术则不一样，开发者可以使用熟悉的编程语言开发软件，之后用容器技术打包构建，便可以一键运行在所有支持该容器技术的平台上。

容器技术具有更快的交付和部署速度，而且相较于其他虚拟化技术，容器技术更加轻量。

1.2 容器技术的前世今生

如果说工业上的集装箱是从一个箱子开始的，那么软件行业上的容器则是从文件系统隔离开始的。

1.2.1 容器技术的起源

最早的容器技术大概是 `chroot`（1979 年）了，它最初是一个 UNIX 操作系统上的系统调用，用于将一个进程及其子进程的根目录改变到文件系统中的一个新位置，让这些进程只能访问到该目录。直到今天，主流 Linux 上还有这个工具。

打开一个终端，输入 `chroot --help`，查看一下这个古老的命令。

```
$ chroot --help
```

```
用法: chroot [选项] 新根 [命令 [参数]...]
```

```
或: chroot 选项
```

```
以指定的新根为运行指定命令时的根目录。
```

```
--userspec=用户:组    指定所用的用户及用户组(可使用“数字”或“名字”)
```

```
--groups=组列表      指定可供选择的用户组列表，形如组 1，组 2，组 3...
```

```
--help              显示此帮助信息并退出
```

```
--version           显示版本信息并退出
```

```
If no command is given, run '$SHELL' -i (default: '/bin/sh -i').
```

```
请向 bug-coreutils@gnu.org 报告 chroot 的错误
```

```
GNU coreutils 的主页: <http://www.gnu.org/software/coreutils/>
```

```
GNU 软件一般性帮助: <http://www.gnu.org/gethelp/>
```

```
要获取完整文档，请运行: info coreutils 'chroot invocation'
```

`chroot` 这个命令主要用来把用户的文件系统根目录切换到指定的目录下，实现简单的文件系统隔离。可以说 `chroot` 的出现是为了提高安全性，但这种技术并不能防御来自其他方面的攻击，黑客依然可以逃离设定访问宿主机上的其他文件。

1.2.2 容器技术的发展

2000 年, 由 R&D Associates 公司的 Derrick T. Woolworth 为 FreeBSD 引入的 FreeBSD Jails 成为了最早的容器技术之一。与 chroot 不同的是, 它可以为文件系统、用户、网络等的隔离增加进程沙盒功能。因此, 它可以为每个 jail 指定 IP 地址, 可以对软件的安装和配置进行定制等。

紧接着出现了 Linux VServer, 这是另外一种 jail 机制, 用于对计算机系统上的资源(如文件系统、CPU 处理时间、网络地址和内存等)进行安全划分。每个所划分的分区叫做一个安全上下文 (Security Context), 在其中的虚拟系统叫做虚拟私有服务器 (Virtual Private Server, VPS)。

在 2004 和 2005 年期间分别出现了 Solaris Containers 和 OpenVZ 技术, 在可控性和便捷性上更胜一筹, 如图 1.3 所示。



图 1.3 常见的容器技术

到了 2006 年, Google 公司公开了 Process Containers 技术, 用于对一组进程进行限制、记账、隔离资源 (CPU、内存、磁盘 I/O、网络等)。后来为了避免和 Linux 内核上下文中的“容器”一词混淆, 而改名为 Control Groups (简称为 Groups)。2007 年被合并到了 Linux 2.6.24 内核中。

在前面 Cgroups 等技术出现以后, 容器技术有了更快的发展。如图 1.4 给出了容器技术的发展史。

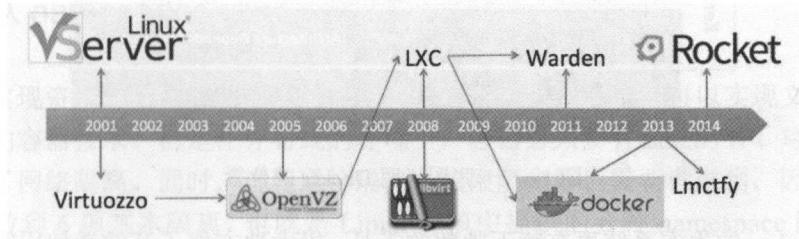


图 1.4 容器技术发展史

2008 年出现了 LXC (Linux Containers), 它是第一个最完善的 Linux 容器管理器的实现方案, 是通过 Cgroups 和 Linux 名字空间 namespace 实现的。LXC 存在于 liblxc 库中, 提供了各种编程语言的 API 实现。与其他容器技术不同的是, LXC 可以工作在普通的 Linux 内核上, 而不需要增加补丁。

LXC 的出现为后面一系列工具的出现奠定了基础。2011 年 CloudFoundry 发布了 Warden。不像 LXC, Warden 并不紧密耦合到 Linux 上, 而是可以工作在任何可以提供隔

离环境的操作系统上。它以后台守护进程的方式运行，为容器管理提供了 API。

在 2013 年，Google 公司发布了 Lmctfy，它是一个 Google 容器技术的开源版本，提供 Linux 应用容器。Google 启动这个项目，旨在提供性能可保证的、高资源利用率的、资源共享的、可超售的、接近零消耗的容器。Lmctfy 首次发布于 2013 年 10 月。到了 2015 年，Google 公司决定贡献其核心的 Lmctfy 概念，并抽象成 libcontainer。现在为 Kubernetes 所用的 cAdvisor 工具就是从 Lmctfy 项目的成果开始的。

libcontainer 项目最初由 Docker 发起，现在已经被移交给了开放容器基金会（Open Container Foundation）。

同年，dotCloud 发布了 Docker（Logo 是一条鲸鱼驮着一堆集装箱，如图 1.5 所示）——至今最流行和使用最广泛的容器管理系统。在 LXC 的基础上，Docker 进一步优化了容器的使用体验，使得容器更容易操作和管理。

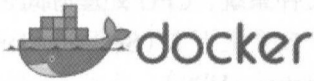


图 1.5 Docker Logo（标志）

Docker 提供了从构建、运行、管理、监控等一系列工具，引入了一整个管理容器的生态系统，包括高效分层的容器镜像模型、全局和本地的容器注册库、清晰的 REST API、命令行等。这是 Docker 与其他容器平台最大的不同。在如图 1.6 中可以看到 Docker 跨越了多个层面，整合了一系列零散的工具从而达到一系列便捷的操作。这是当时 Docker 从众多容器技术中脱颖而出的一个重要原因。

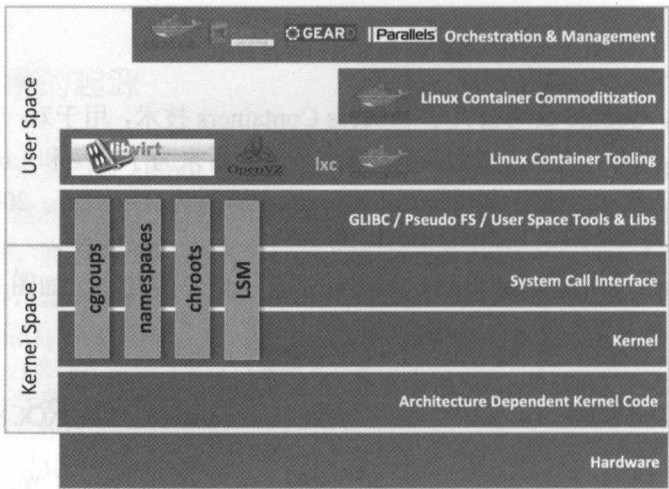


图 1.6 一张图说明 Docker 的位置

围绕 Docker 的生态系统更有数不胜数的工具，极大地方便了开发者使用容器技术。关于 Docker 的更多特性，将在第 2 章介绍。

Docker 开始阶段使用的也是 LXC，之后采用自己开发的 libcontainer 替代了 LXC。

之后出于各种原因，CoreOS 启动项目 Rocket，非常类似于 Docker，但是修复了一些 Docker 中发现的问题。与 Docker 相比，Rocket 是在一个更加开放的标准 App Container 规范上实现的。现今不少容器管理工具都支持 Rocket 与 Docker。

2015 年微软公司也在 Windows Server 上为其基于 Windows 的应用添加了容器支持，称之为 Windows Containers，与 Windows Server 2016 一同发布。通过该实现，Docker 可以

原生地在 Windows 上运行 Docker 容器，而不需要启动一个虚拟机来运行 Docker（Windows 上早期运行 Docker 需要使用 Linux 虚拟机）。同年，Mac OS 也原生支持运行 Docker 容器。如图 1.7 所示为官网给出的下载按钮。至此 Docker 完成了三大平台的适配。

容器虚拟化技术经过几十年不断的发展与完善，相继加入了 pivot_root 等很多技术。市场上也出现了一些商业化的容器技术公司。在这些公司与全球开发者的共同努力下，容器技术得到不断推进和发展。最后核心容器技术进入了 Linux 的内核主线，再后来诸多大厂加入开发的 libcontainer，使得如今人人皆可得心应手地操作容器。

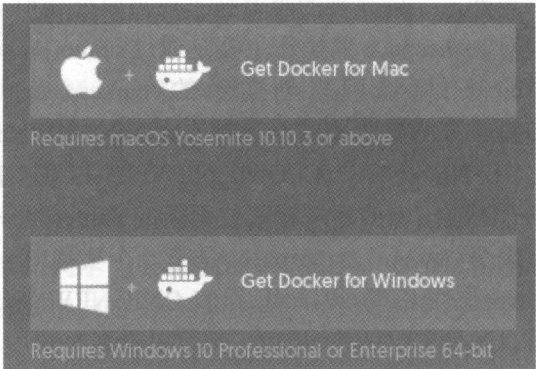


图 1.7 Docker 在 2016 年正式支持 Windows 10 与 Mac OS

1.3 容器的原理

前文提到，容器的核心技术是 Cgroups 与 namespace，在此基础上还有一些其他工具共同构成容器技术。容器本质上是宿主机上的进程。容器技术通过 namespace 实现资源隔离，通过 Cgroups 实现资源控制，通过 rootfs 实现文件系统隔离，再加上容器引擎自身的特性来管理容器的生命周期。

简单来说，本书所说的 Docker 的早期其实就相当于 LXC 的管理引擎，LXC 是 Cgroups 的管理工具，Cgroups 是 namespace 的用户空间管理接口。namespace 是 Linux 内核在 task_struct 中对进程组管理的基础机制。

1.3.1 从 namespace 说起

想要实现资源隔离，第一个想到的就是 chroot 命令。通过它可以实现文件系统隔离，这是最早的容器技术。但是在分布式的环境下，容器必须要有独立的 IP、端口和路由等，自然就有了网络隔离。同时，进程通信隔离、权限隔离等也需要考虑到，因此基本上一个容器需要做到 6 项基本隔离，也就是 Linux 内核中提供的 6 种 namespace 隔离，如表 1.1 所示。

表 1.1 namespace 说明

namespace	隔离内容
IPC	信号量、消息队列和共享内存
Network	网络资源
Mount	文件系统挂载点
PID	进程 ID
UTS	主机名和域名
User	用户 ID 和组 ID

当然，一项完善的容器技术还需要处理很多工作。

对 namespace 的操作主要是通过 clone()、setns()、unshare()这 3 个系统调用来完成的。

clone() 可以用来创建新的 namespace。clone() 有一个 flags 参数，该参数以 CLONE_NEW* 为格式，包括 CLONE_NEWNS、CLONE_NEWIPC、CLONE_NEWUTS、CLONE_NEWNET、CLONE_NEWPID 和 CLONE_NEWUSER，通过传入这些参数后，由 clone() 创建出来的新进程就位于新的 namespace 之中了。

因为 Mount namespace 是第一个实现的 namespace，当初实现没有考虑到还有其他 namespace 的出现，因此用了 CLONE_NEWNS 的名字，而不是 CLONE_NEWMNT 之类的名字。其他 CLONE_NEW* 都可以看名字知用途。

那么，如何为已有的进程创建新的 namespace 呢？这就需要用到 unshare() 了，使用 unshare() 调用的进程会被放进新的 namespace 里面。而 setns() 则是将进程放到已有的 namespace 中，docker exec 命令的实现原理就是 setns()。

事实上，开发 namespace 的主要目的之一就是实现轻量级虚拟化服务，在同一个 namespace 下的进程可以彼此响应，而对外界进程隔离，这样在一个 namespace 下，进程仿佛处于一个独立的系统环境中，以达到容器的目的。

上面介绍的是一些概念，下面来实践一下。因为 user namespace 是在 Linux 内核 3.8 之后才支持的，所以本节讨论的 namespace 均是 3.8 以后的版本。

1. 查看当前进程的namespace

在了解 namespace API 之前，先来了解如何查看进程的 namespace。在 root 用户模式下执行：

```
# ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 root root 0 6月 10 20:29 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 6月 10 20:29 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 6月 10 20:29 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 6月 10 20:29 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 6月 10 20:29 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 6月 10 20:29 uts -> uts:[4026531838]
```

这里的\$\$是指当前进程 ID 号。可以看到诸如 4026531839 这样的数字，表示当前进程指向的 namespace。当两个进程指向同一串数字时，表示它们处于同一个 namespace 下。

2. 使用clone()创建新的namespace

创建一个 namespace 的方法是使用 clone() 系统调用，它会创建一个新的进程。为了说明创建的过程，给出 clone() 的原型如下：

```
int clone(int(*child_func)(void*), void*child_stack, int flags, void*arg);
```

本质上，clone() 是一个通用的 fork() 版本。fork() 的功能由 flags 参数控制。总的来说，约有超过 20 个不同的 CLONE_* 标志控制 clone() 提供不同的功能，包括父子进程是否共享如虚拟内存、打开的文件描述符和子进程等资源。如果调用 clone() 时设置了一个 CLONE_NEW* 标志，一个与之对应的新的命名空间将被创建，新的进程属于该命名空间。可以使用多个 CLONE_NEW* 标志的组合。

3. 使用setns()关联一个已经存在的namespace

当一个 namespace 没有进程时还保持其打开，这么做是为了后续添加进程到该 namespace。而添加这个功能就是使用 setns()系统调用来完成，这使得调用的进程能够和 namespace 关联，docker exec 就需要用到这个方法：

```
int setns(int fd, int nstype);
```

- fd 参数指明了关联的 namespace，其指向了 /proc/PID/ns 目录下一个符号链接的文件描述符。可以通过打开这些符号链接指向的文件或者打开一个绑定到符号链接的文件来获得文件描述符。
- nstype 参数运行调用者检查 fd 指向的命名空间的类型，如果这个参数等于数，将不会检查。当调用者已经知道 namespace 的类型时这会很有用。当 nstype 被赋值为 CLONE_NEW* 的常量时，内核会检查 fd 指向的 namespace 的类型。

要把 namespace 利用起来，还要使用 execve()函数（或者其他的 exec()函数），使得我们能够构建一个简单但是有用的工具，该函数可以执行用户命令。

4. 使用unshare()在已有进程上进行namespace隔离

unshare()和 clone()有些像，不同的地方是前者运行在原有进程上，相当于跳出原来 namespace 操作，Linux 自带的 unshare()就是通过调用 unshare()这个 API 来实现的。

```
$ unshare
Usage:
unshare [options] <program> [args...]
-h, --help          usage information (this)
-m, --mount          unshare mounts namespace
-u, --uts            unshare UTS namespace (hostname etc)
-i, --ipc            unshare System V IPC namespace
-n, --net            unshare network namespace
For more information see unshare(1).
```

由于 Docker 没有使用这个系统调用，所以不展开。除此之外，像 fork()这样的函数也可以实现 namespace 隔离，但并不属于 namespace API 的一部分。有兴趣的读者可以扫描下方二维码阅读相关资料。



1.3.2 认识 Cgroups

Cgroups 是 Linux 内核提供了一种可以限制、记录、隔离进程组（process groups）所使用的物理资源（如 CPU，内存，I/O 等）的机制。最初由 Google 公司的工程师提出，后来

被整合进 Linux 内核。Cgroups 也是 LXC 为实现虚拟化所使用的资源管理手段，可以说没有 Cgroups 就没有 LXC。

目前，Cgroups 有一套进程分组框架，不同资源由不同子系统控制。一个子系统就是一个资源控制器，比如 CPU 子系统就是控制 CPU 时间分配的一个控制器。子系统必须附加（attach）到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群（control groups）都受到这个子系统的控制。

Croups 各个子系统作用如下。

- Blkio: 为块设备设定输入/输出限制，比如物理设备（磁盘、固态硬盘、USB 等）。
- Cpu: 提供对 CPU 的 Cgroups 任务访问。
- Cpuacct: 生成 Cgroups 中任务所使用的 CPU 报告。
- Cpuset: 为 Cgroups 中的任务分配独立 CPU（在多核系统）和内存节点。
- Devices: 允许或者拒绝 Cgroups 中的任务访问设备。
- Freezer: 挂起或者恢复 Cgroups 中的任务。
- Memory: 设定 Cgroups 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。
- Net_cls: 使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（tc）识别从具体 Cgroup 中生成的数据包。
- Net_prio: 设置进程的网络流量优先级。
- Huge_tlb: 限制 HugeTLB 的使用。
- Perf_event: 允许 Perf 工具基于 Cgroups 分组做性能监测。

这样说理解起来也很吃力，下面就通过命令来挂载 Cgroups。

```
# mount -t cgroup -o cpuset cpuset /sys/fs/cgroup/cpuset
```

这个动作一般情况下已经在 Linux 启动的时候做了。

查看 Cgroupfs:

```
# cpuset ls
cgroup.clone_children      cpuset.memory_pressure_enabled
cgroup.procs               cpuset.memory_spread_page
cgroup.sane_behavior       cpuset.memory_spread_slab
cpuset.cpu_exclusive       cpuset.mems
cpuset.cpus                cpuset.sched_load_balance
cpuset.effective_cpus      cpuset.sched_relax_domain_level
cpuset.effective_mems      docker
cpuset.mem_exclusive       notify_on_release
cpuset.mem_hardwall        release_agent
cpuset.memory_migrate      tasks
cpuset.memory_pressure
```

在主流 Linux 发行版下，都可以通过/etc/cgconfig.conf 或者 cgroup-bin 的相关指令来配置 Cgroups。

```
mount {
    cpuset = /sys/fs/cgroup/cpuset;
    momory = /sys/fs/cgroup/momory;
}
group cnsworder/test {
    perm {
        task {
            uid = root;
```

```
        gid = root;
    }
    admin {
        uid = root;
        gid = root;
    }
}
cpu {
    cpu.shares = 1000;
}
}
```

然后通过命令行把一个进程移动到这个 Cgroups 之中。

```
# mount -t group -o cpu cpu /sys/fs/cgroup/cpuset
# cgcreate -g cpu,memory:/cnsworder
# chown root:root /sys/fs/cgroup/cpuset/cnsworder/test/*
# chown root:root /sys/fs/cgroup/cpuset/cnsworder/test/task
# cgrun -g cpu,memory:/cnsworder/test bash
```

关于 Cgroups 子系统，本书不再过多讲述，读者可以扫描以下二维码找到很不错的学习资料，了解更多内容。

Acch Linux Wiki	Linux 内核文档 V1	Linux 内核文档 V2	Fedora 文档
			

1.3.3 容器的创建

前面只是非常简单地介绍了 namespace 和 Cgroups 两个概念。实际上各个 namespace 的具体介绍与各个 Cgroups 子系统的介绍都没有深入讲解到，但通过前面两节的学习，相信读者已经大致有了容器创建过程的雏形。

(1) 系统调用 clone() 创建新进程，拥有自己的 namespace。

该进程拥有自己的 pid、mount、user、net、ipc 和 uts namespace。

```
# pid = clone(fun, stack, flags, clone_arg);
```

(2) 将 pid 写入 Cgroup 子系统这样就受到 Cgroups 子系统控制。

```
# echo$pid >/sys/fs/cgroup/cpu/tasks
# echo$pid >/sys/fs/cgroup/cpuset/tasks
# echo$pid >/sys/fs/cgroup/bikio/tasks
# echo$pid >/sys/fs/cgroup/memory/tasks
# echo$pid >/sys/fs/cgroup/devices/tasks
# echo$pid >/sys/fs/cgroup/feezer/tasks
```

(3) 通过 pivot_root 系统调用，使进程进入一个新的 rootfs，之后通过 exec() 系统调用，在新的 namespace、Cgroups、rootfs 中执行/bin/bash。

```
fun() {
    pivot_root("path_of_rootfs/", path);
```

```
    exec("/bin/bash");  
}
```

通过上面的操作，成功地在容器中运行了/bin/bash。

1.4 容器云

每一项技术成熟后都会衍生出一系列技术，例如当 Docker 推开容器世界的大门时，围绕容器技术的生态系统迅速发展起来。无论是个人还是企业，在使用上都有各种各样的需求，例如跨主机连接容器，各种类型的负载均衡，持续构建、集成和交付，以及大规模容器管理等。

虽然 Docker 提供了较为便捷的操作方式，但是在开发、生产环境中，网络、存储、集群和高可用等问题层出不穷。仅凭 Docker 是无法做到面面俱到的。于是从容器到容器云就成了容器技术的必然发展路径。

国内现在以 Docker 容器云为“卖点”的初创公司不在百家之下，国外更是不用言说。可见围绕 Docker 容器云还有很多需要开发者去完善的地方。如图 1.8 展示了目前 Docker 的生态圈的一部分。可以看到这些工具围绕 Docker 进行扩展补充，已经形成了非常发达的生态系统网络。

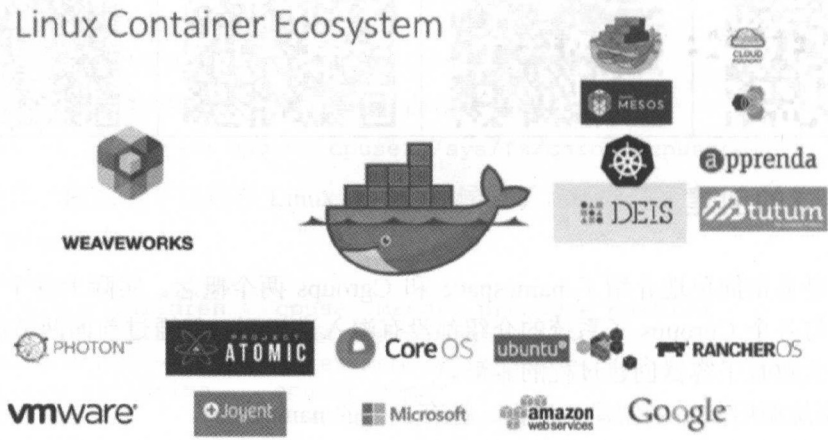


图 1.8 容器技术生态圈

完整来说，容器云是以容器为资源分割和调度的基本单位，通过容器封装软件运行环境，为用户提供一个集构建、发布和运行于一体的分布式应用平台。它与 IaaS、PaaS 等不同，容器云可以共享与隔离资源、编排与部署容器。在这一点上容器云与 IaaS 相似。但是容器云也可以渗透到应用支撑与运行时环境，在这一点上与 PaaS 类似。

当然容器云并不是特指以 Docker 为基础的容器技术。使用其他容器技术（CoreOS 的 Rocket 项目）实现容器云也是可以的。

后来的微服务（Microservices）和 Serverless 可以说是在容器技术基础上的突破性发展。微服务 Microservices 在软件架构上可以将容器用于部署。微服务并不是一个新东西，只是一个相比标准的 Web 服务超快的轻量级 Web 服务。这是通过将功能单元（也许是一个单一

服务或 API 方法) 打包到一个服务中, 并内嵌其到一个轻量级 Web 服务器软件中实现的。Docker 与微服务的联手可以说打开了又一扇大门。

1.5 容器与 Docker

关于容器是否是 Docker 的核心技术在业内一直存在争议。Docker 的核心是对分层镜像的创新使用, 还是统一了应用的打包分发和部署方式, 一直没有定论。之所以有这样的争议, 是因为 Docker 的创新不一定要依赖容器技术, 像基于传统的 hypervisor 也可以做到。

而且官方对 Docker 核心技术功能的描述 “Build, Ship and Run” 中也确实没有体现与容器相关的内容。但是毫无疑问, 容器成就了 Docker, 而 Docker 也极大地促进了容器技术的发展。

实际上, 从 Docker 公司的表现来看, 它不会单纯地只是做一个 CaaS (容器即服务) 服务商。当前的 Docker 更像是一艘巨轮, 野心其实不小, 如图 1.9 所示。这也是 Rocket 项目诞生的一个重要原因。至于 Docker 能否引领容器领域或者是另有枭雄杀出一条血路, 我们不妨拭目以待。

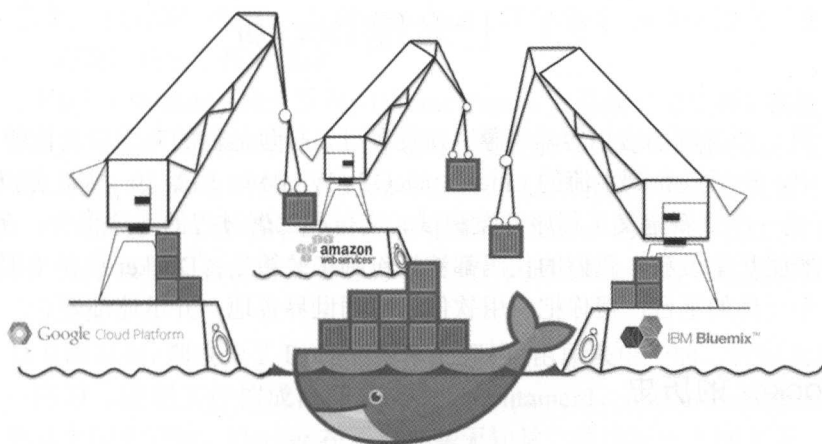


图 1.9 当前 Docker 就像是一艘巨轮

1.6 本章小结

通过本章的学习, 读者对容器的基本概念有了大致了解, 对容器的运行原理和调用方法有了基本认识, 也对容器技术的历史有了认识, 了解了 Docker 在容器技术中的位置。第 2 章将揭开 Docker 的神秘面纱, 认识这位出身 “寒门” 的 “灰姑娘” 是如何成为万众瞩目的 “公主” 的。

第 2 章 Docker 简介

第 1 章已经大致了解了容器技术，在本章节中，我们将进一步认识 Docker 的架构，以及了解 Docker 与其他容器技术的区别。

从本章开始将进入本书的重点，本章先了解 Docker 的架构，主要包括：

- Docker 定义。
- Docker 的优缺点与目的。
- Docker 与其他容器技术对比。
- Docker 与虚拟机对比。
- Docker 基本架构。

2.1 什么是 Docker

Docker 是一个开源的应用容器引擎，开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到主流的 Linux / macOS / Windows 机器上，实现虚拟化。

Docker 是一个重新定义了程序开发测试、交付和部署过程的开放平台。在 Docker 的世界里，容器就是集装箱，我们的代码都被打包到集装箱里；Docker 就是集船坞、货轮、装卸、搬运于一体的平台，帮你把应用软件运输到世界各地，并迅速部署。

2.1.1 Docker 的历史

2010 年，几个年轻人在旧金山成立了一个叫做 PaaS 平台的公司，起名为 dotCloud。什么是 PaaS 呢？PaaS 的全称是 Platform as a Service，也就是平台即服务。

传统的软件产品开发过程一般是这样的：

- (1) 制定产品定位和需求。
- (2) 制作界面原型。
- (3) 搭建开发环境和技术栈。
- (4) 构建基础技术框架和服务。
- (5) 构建测试环境。
- (6) 实现产品功能。
- (7) 迭代开发/测试。

但是如果使用的是 PaaS 平台，可以直接忽略 (3)、(4)、(5) 这 3 个步骤。无论选择哪个技术栈，PaaS 都会为你提供相关的配套设置，包括语言环境、运行环境、存储和各种基础服务。

dotCloud 就是一个比较完善的 PaaS 平台。它把需要花费大量时间的手工工作和重复劳动抽象成组件和服务，并放到了云端，还提供了各种监控、警告和控制功能，方便开发者管理和监控自己的产品。

但是事实上开发者对开发环境和开发工具非常敏感并要求极高。PaaS 的概念虽好，但是由于认知、理念和技术的局限性，市场的接受度并不高，市场的规模也不够大。此外，IBM、微软、Amazon、Google、VMware 等巨头相继进场，可谓强敌林立，在这种情况下，dotCloud 可以说举步维艰。

于是 dotCloud 决定把核心引擎开源，这个基于 Linux Container 技术的核心管理引擎一经开源立刻引起业界广泛关注。这个容器管理引擎大大降低了容器技术的使用门槛，轻量级、可移植、虚拟化、与语言无关，写了程序放上去做成镜像可以随处部署和运行，开发、测试和生产环境统一了，还能进行资源管控和虚拟化，就连众多巨头们也纷纷表示要接入或支持这个引擎。这个引擎就是 Docker，用 Go 语言写成。

2013 年 10 月 dotCloud 公司更名为 Docker 股份有限公司，2014 年 8 月 Docker 宣布把 PaaS 的业务 dotCloud 出售给位于德国柏林的平台即服务提供商 cloudControl，使得 Docker 公司集中更多的精力放到了 Docker 相关的研发上。

在随后的几年里，Docker 快速发展成为了容器领域的“领头羊”。从 Docker 1.9.0 版本（2015 年）开始，执行驱动默认改为 libcontainer，这意味着 Docker 迈向了更广阔的舞台，也为 2016 年的发展奠定了技术基础。

时间迈入 2016 年，Docker 1.10.0 发布，Docker Engine 支持配置热更新，容器与 Docker Daemon 的耦合性大大降低。该版本 Docker 还划时代地支持了 User namespace 与 seccomp，安全性极大提高，随着 Registry 升级，Docker 的安全性已经十分接近虚拟机技术。

与此同时，传统的容器管理工具 LXC 也开始退出 Docker 的舞台，LXC 伴随 Docker 接近三年的时间，终于被更完善的容器管理方案取代，同时 Docker 的飞速发展以及追求卓越的野心也一览无余。

Docker 1.11.0 的发布，则改变了 Docker 原来的架构：由原来单一的二进制文件 docker，拆分为 4 个不同的二进制文件构成，即 docker、containerd、docker-containerd-shim 和 docker-runc。从这个版本开始，Docker 在用户“毫无知觉”的情况下全面升级，成为首个完全兼容 OCI（开放式容器协议）标准的运行时。该版本之后，Docker Engine 完全基于 runC 和 containerd。

Docker Engine 负责镜像管理，containerd 管理容器，包括容器的启动、停止、暂停和销毁。Docker Engine 将镜像交付到 containerd 运行，containerd 则使用 runC 来运行容器。鉴于容器运行时与引擎隔离，Engine 能够重启和升级，无须重启容器。

此后又是一个划时代版本 Docker 1.12，从这个版本开始，Docker 原生支持编排功能，Swarm 集群蜕变为 SwarmKit，融入 Docker Engine，内置负载均衡功能，还实现新的 plugin 命令来管理各种插件等。

总结 Docker 四年发展历程，不难发现 Docker 的野心。第一年专注软件构建，对接构建下游，营造镜像生态，良好的体验很快吸引了大批开发者；第二年瞄准服务容器管理，发布调度平台，打造交付流程，同时收紧了部分功能，加强控制容器市场；第三年大力整合企业资源，完善平台功能，着手应用编排，推出面向企业的 Docker Cloud 平台；到了第四年，在 DevOps 理念下，Docker 一方面往上原生集成编排，挤掉 Kubernetes 和 Mesos，

另一方面往下，迈向 OS 化，发布 libnetwork 强化网络管理，排除第三方工具。
更多历史趣闻，读者可以扫描下方二维码继续阅读。



2.1.2 Docker 的现状

截至 2016 年 08 月，Docker 已经成长为云计算相关领域最受欢迎的开源项目，Amazon、Google、IBM、Microsoft、Red Hat 和 VMware 分别表示已经支持 Docker 技术。

有 Linux 的地方，就可以运行 Docker，2016 年 6 月 Docker 更是登录了 Mac OS 与 Windows 平台。

如图 2.1 所示，截至 2016 年 10 月 1 日，Docker 在 Github 共有 1507 名开发者参与到了开发中，提交了接近三万次的 commit。Docker 的生态圈也非常活跃，在 Github 公开项目中，至少有 11 万个项目与 Docker 相关。

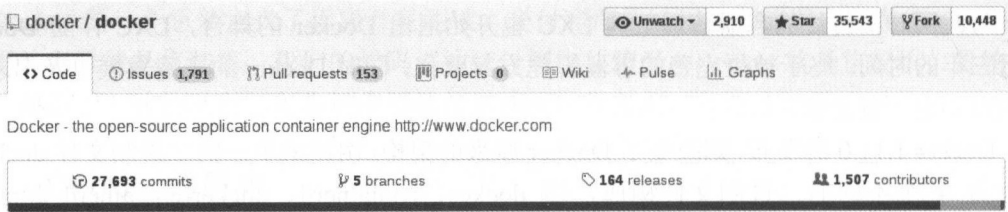


图 2.1 Docker 项目在 Github 上十分活跃

如图 2.2 所示，在 Docker 2016 年度报告中，使用 Docker 的用户约九成是进行应用开发，接近八成是网页应用。从调查看，用户在开发和运维过程中获益最多，这也体现了 Docker 的理念。

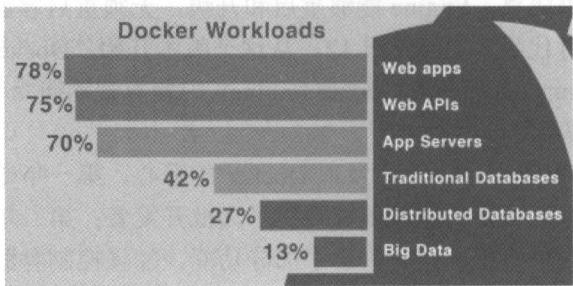


图 2.2 78%是网页应用

如图 2.3 所示，在集成了 Swarm 之后的新版本 Docker 中，Docker 启动一个新容器比 Kubernetes 快 5 倍，遍历所有容器比 Kubernetes 快 7 倍。

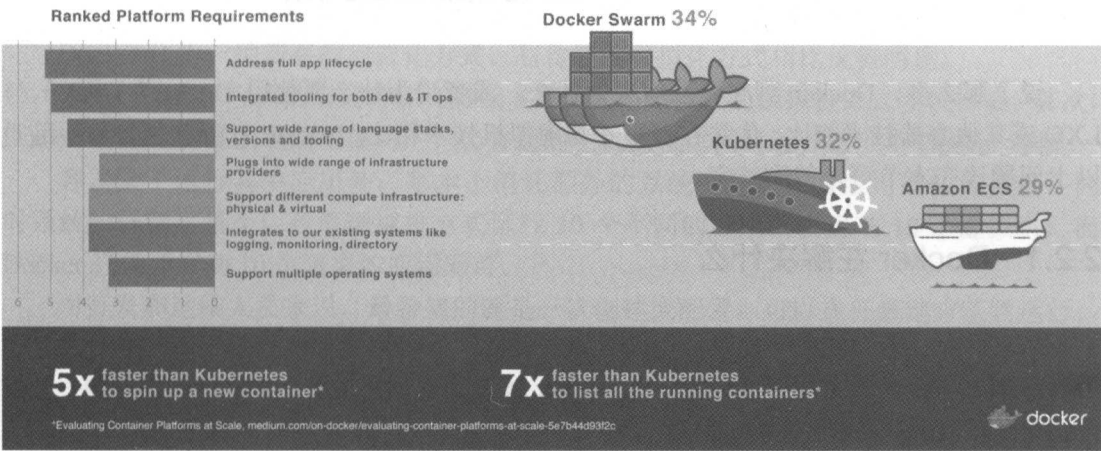


图 2.3 Docker 与 Kubernetes 比较

2.1.3 Docker 的未来

Docker 公司目前发力的三大方向都有了长足进步，即发展核心能力（libcontainer）、跨业务管理（libswarm）和容器间消息（libchan）。与此同时，通过收购 Orchard labs，Docker 公司表达了利用自身生态系统的意愿。但是，这不能仅仅关注 Docker 公司，这个项目的贡献者还来自于一些大牌公司，如谷歌、IBM 和 Red Hat。在 Docker 首席技术官 Solomon Hykes 的掌舵下，Docker 公司和 Docker 项目的技术领先有着明确的联系。在项目初始的 18 个月里，它已经显示出通过自己的输出来快速前进的能力，并且没有减弱的迹象。

许多围绕 Docker 的创业公司试图找出已经由虚拟机普及而驱动的企业预期和现有 Docker 生态系统之间的差距（和机会）。在网络存储和细粒度的版本管理（用于容器中的内容）领域，现有 Docker 生态系统做得并不好，这就为初创企业和在职人员提供了机会。

随着时间的推移，虚拟机和容器（Docker 中的“运行（run）”部分）之间的区别很可能变得不再那么重要，这将使注意力转到“构建（build）”“交付（ship）”方面。

值得注意的是，Docker 公司绝对不会只想做一个引擎组件，集成 Swarm 可以说是 Docker 向 Google 这些巨头发出的挑战，Docker 公司更倾向于平台化，企图打造工程技术界的“软件市场”。在这一点上，相信其他竞争者不会妥协，所以 Docker 容器技术的未来将与 Docker 公司的未来紧密关联。可见的未来里，Docker 公司的策略依旧会是破坏向后的兼容性，因为其对 Docker 引擎本身的定位就是产品而非社区自有的服务组件。但是，不管结局如何，有一点可以肯定，那就是 Docker 必然给软件行业带来一次不寻常的革命，也许 Docker 不是笑到最后的赢家，但至少它是目前可见的最有力的竞争者。

2.2 Docker 的功能及优缺点

说了那么多，Docker 到底可以用来做什么，或者说为什么要使用 Docker？Docker 与 LXC 或其他容器技术相比，优势在哪里？与虚拟机技术相比，Docker 有哪些优势与不足？以上问题读完本节可以得到答案。

2.2.1 Docker 在解决什么

从表面看，最直观的一点是，以前开发软件时，各种依赖环境都要考虑到，开发好之后迁移到另一台机器上却不能运行，而 Docker 解决了运行环境和配置、依赖等问题，使软件发布迁移容易了许多。

第二点就是更轻量的虚拟化，节省了虚拟机的性能损耗，却得到了和虚拟机差不多的隔离环境。

第三点，用第 1 章提到的集装箱做比喻，在一艘大船上，集装箱可以把货物规整地摆放起来并且各种各样的货物被集装箱标准化了，集装箱和集装箱之间不会互相影响，因此就不需要专门运送水果的船和专门运送工业品的船了。只要这些货物在集装箱里封装好，那么就可以用一艘大船把它们都运走。在这一点上，Docker 还解决了软件调度分发的问题。

第四点，更高的资源利用率。服务自身和依赖环境对资源的浪费，以及服务没有利用到的空闲资源往往是资源利用率低下的罪魁祸首。现实情况下资源被拆分成一台台计算机，服务也要跟着拆分。当服务的拆分不能保证和资源拆分相同的粒度时，空闲资源就产生了。

服务拆分的粒度越粗，与资源不匹配的矛盾会越激烈；拆分粒度越细，带来的资源浪费也就越多。另一方面，服务内聚提高会带来更高的耦合度和风险，反之会带来更高的部署和维护成本等。而 Docker 结合微服务带来了一个希望，以极小的代价换取极大的资源利用率。

总结起来就是，Docker 带来了更高效的服务部署、启动方式、简化配置，在容器中开发完成后快速部署于各主流系统，解决了依赖问题；对 CPU、内存、网络和文件系统的隔离使其成为可替代虚拟机技术的选项之一，机器资源利用率的提高，不需要为虚拟一个环境而耗费大量资源。

由于 Docker 本身带来了这两点好处，通过 Docker 和相关的生态系统，可以更简单地实现对系统的监控，把管理对象由“机器”改为抽象的“资源”，基于对资源的抽象，提供更灵活的服务部署、调度机制。这也改变了传统应用交付的方式，软件的开发和管理从部门间的装配和调试转换为部件的简单替换。软件的管理和共享从代码层次向应用层次上升，给了我们更多的选择自由性，使软件架构更加灵活。

其实，与其纠结于 Docker 在解决什么，不如拆分成两个问题：我们要做什么？Docker 能帮我们做什么？

2.2.2 为什么选择 Docker

Docker 作为一种新兴的虚拟化方式，与传统的虚拟化方式相比优势明显。

首先，Docker 容器的启动可以在秒级实现，这相比传统的虚拟机方式要快得多。其次，Docker 对系统资源的利用率很高，一台普通的主机上可以同时运行数千个 Docker 容器。

容器除了运行其中应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。传统虚拟机方式运行 10 个不同的应用就要启动 10 个虚拟机，而 Docker 只需要启动 10 个隔离的应用即可。

对开发和运维人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。开发人员使用一个标准的镜像来构建一套开发容器，开发完成之后，运维人员可以直接使用这个容器来部署代码。

而这正是 Docker 可以做到的，Docker 容器的运行不需要额外的 hypervisor 支持，它是内核级的虚拟化，因此可以实现更高的性能和效率。

Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人计算机、服务器等。这种兼容性可以让用户把一个应用程序从一个平台直接迁移到另外一个平台上。使用 Docker，只需要小小的修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的管理。

2.2.3 Docker 的缺点

Docker 虽然日趋完善，但在某些方面依旧有着很明显的短板。最大的问题是安全性，Docker 原来使用 LXC 作为默认的执行引擎，从 Docker0.9 之后采用 libcontainer 作为默认的执行引擎。Docker 需要一个 root 权限的守护进程一直在主机上运行，不可避免会出现安全漏洞。

Docker 对 namespace 过分信任，而 namespace 的攻击面比一般的 Hypervisor 要大，Xen 在 Linux 里面有 129 个 CVEs (Common Vulnerabilities and Exposures)，与之相比它却有 1279 个。当然，这在某些情况下也是可以接受的，如在 Travis CI 里面以公有的方式来构建，但是在私有情况下和多用户的环境下，就显得比较危险了。

除此之外，Docker 的网络一直是广大开发者不满的一个地方，尽管 Docker 在网络管理方面做了很多努力，但是容器技术的网络问题由来已久，这方面的薄弱不是 Docker 一己之力就能解决的。

2.3 Docker 和虚拟机

Docker 与虚拟机经常被拿来作比较，相信在第 1 章中对容器技术的介绍已经使读者对容器的概念有了基本认识。相信读者也接触过虚拟机，两者的区别和关系，个人觉得并非是竞争，而是互补。

2.3.1 Docker 与虚拟机的区别

虚拟机和 Docker 最明显的差别是虚拟机需要安装操作系统（安装 Guest OS）才能执行应用程序，而 Docker 内不需要安装操作系统。Docker 技术不是在 OS 外来建立虚拟环境，而是在 OS 内的核心系统层来打造虚拟执行环境，通过共享宿主机 OS 的做法，取代一个 Guest OS 的功用。Docker 也因此被称为是操作系统虚拟化技术。

如图 2.4 所示，因为 Docker 技术采取共享宿主机 OS 的做法，而不需在每一个 Docker 容器内执行 Guest OS，因此建立 Docker 容器不需要等待操作系统开机时间，也不需要加载操作系统的额外进程。

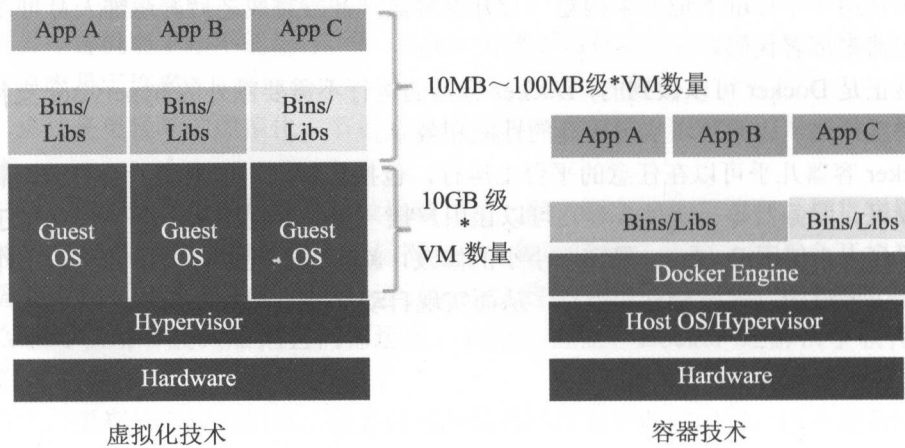


图 2.4 容器技术与虚拟机技术对比

虚拟机（Virtual Machine）从操作系统层下手，目标是建立一个可以用来执行整套操作系统的沙盒独立执行环境。而 Docker 则是直接将一个应用程序所需的相关程序代码、函数库、环境配置文件都打包起来建立沙盒执行环境。

从隔离上来讲，虚拟机是用来进行硬件资源划分的完美解决方案，它利用了硬件虚拟化技术，通过一个 Hypervisor 层来实现对资源的彻底隔离；而容器则是操作系统级别的虚拟化，利用内核的特性实现，不需要外部辅助。

2.3.2 Docker 与虚拟机的优缺点

Docker 虽然在很多方面优于虚拟机，但是作为一项传统的虚拟技术——虚拟机并非在 Docker 面前一无是处。

如表 2.1 中总结了使用 Docker 容器技术与传统虚拟机技术的特性比较。

表 2.1 Docker 容器技术与虚拟机技术对比

特 性	Docker 容器技术	虚拟机技术
占用磁盘空间	小，甚至只有几十 KB（镜像层的情况）	非常大，可达 GB
启动速度	快，几秒钟	慢，几分钟

(续)

特 性	Docker容器技术	虚拟机技术
运行形态	直接运行于宿主机的内核上，不同容器共享同一个Linux内核	运行于Hypervisor上
并发性	一台宿主机可以启动成百上千个容器	最多几十个虚拟机
性能	接近宿主机本地进程	逊于宿主机
资源利用率	高	低
隔离性	安全隔离	完全隔离

在某些地方，例如 Docker 还无法胜任的地方，虚拟机还会继续发光发热，也许有一天容器会成为最流行的虚拟化技术，但不管如何虚拟机技术都不会消失。

2.4 Docker 与 runC

在 2013 年 Docker 刚发布的时候，它是基于 LXC 的开源容器管理引擎。Docker 把 LXC 复杂的容器创建与使用方式简化为自己的一套命令体系。

随着 Docker 的发展，原有的 LXC 不能满足 Docker 的需求(比如跨平台)，于是 Docker 公司把底层实现都抽象化，底层容器的实现方式变成了一种可变的方案，这就是 libcontainer 的诞生。从此之后，无论是使用 namespace、Cgroups 技术或是使用 systemd 等其他方案，只要实现了 libcontainer 定义的一组接口，Docker 都可以运行。

2.4.1 libcontainer 与 runC

在 DockerCon 2015 期间，Docker 牵头成立了 OCI (Open Container Initiative 开放容器计划) 组织，这个组织的目的是建立起一个围绕容器的通用标准。

容器格式标准是一种不受上层结构绑定的协议，即不限于某种特定操作系统、硬件、CPU 架构、公有云等，这样做的目的是减少因为行业内的恶性竞争，提供一个标准，允许任何人在遵循该标准的情况下开发应用容器技术，这使得容器技术有了一个更广阔的发展空间，OCI 下的容器技术不属于任何公司或个人。

早期 libcontainer 是 Docker 公司控制的一个开源项目，随着 OCI 的成立，Docker 把 libcontainer 项目移交给了 OCI 组织，目前可以在 <https://github.com/opencontainers/runc> 中查看 libcontainer 的源代码，目前 libcontainer 作为 runC 项目的一个子项目。

runC 就是在 libcontainer 的基础上进化而来，Docker 已经表明未来会使用 runC 替代 libcontainer 作为容器 runtime 的工具。

从名字上可以看出 runC 是一个 runtime 工具，而 libcontainer 只是一个 lib 库，不是 runtime 管理工具。runC 通过调用 libcontainer 提供的接口来管理容器。早期 runC 没有出来之前，libcontainer 有一个内置的小工具 nsinit，可以用这个内置小工具来管理容器。后来 runC 基于这个小工具做了改动，并改名为 runC。随着 runC 的不断发展，目前 runC 已经成为一个功能强大的 runtime 工具。当前，我们只需要知道 runC 的核心依旧是 libcontainer

就可以了。

2.4.2 runC 的使用

前面说过容器是提供一个与宿主机系统共享内核但与系统中的其他进程资源相隔离的执行环境。runC 通过调用 libcontainer 包对 namespaces、Cgroups、capabilities 以及文件系统的管理和分配来“隔离”出一个上述执行环境，相当于一个去除了如镜像、Volume 等高级特性的“简化版”Docker，runC 以最朴素简洁的方式达到符合 OCF 标准的容器管理实现。

因为 Docker 是按照 OCF（开放容器格式）开发的，所以 runC 可以读取运行 Docker 的容器。

runC 运行时需要 rootfs，最简单的就是本地已经安装好了 Docker，通过 docker pull 下载一个基本的镜像：

```
$ docker pull busybox
```

使用 docker create 创建一个容器再使用 docker export 导出容器：

```
$ docker export$(docker create busybox) > busybox.tar
```

接下来解压到 rootfs 目录：

```
$ mkdir rootfs
$ tar -C rootfs -xf busybox.tar
```

这样就可以用 runC 来启动一个基于 OCF 的容器了（这里 runC 并不依赖 Docker，使用 Docker 只是为了方便建立一个 rootfs）。

一个 OCF 容器应该包含 config.json 和 runtime.json 以及 rootfs 三大部分，所以还需要用 runc spec 命令来生成一份配置文件：

```
$ runc spec
```

 **注意：**如果还没有安装 runC，那么需要按照如下步骤安装。

```
// 先要在 GOPATH/src 目录下创建一个文件夹名为 'github.com/opencontainers' 。
$ cd github.com/opencontainers
$ git clone https://github.com/opencontainers/runc
$ cd runc
$ make
$ sudo make install
```

完成上述步骤，就可以使用 runC 启动一个容器了：

```
$ runc start
```

2.4.3 runC 原理

前面提到，runC 就是 libcontainer 加上 cli。（cli 的开发包也是托管在 Github 上面的一个开源项目，地址为 github.com/codegangsta/cli）。

libcontainer 是 Docker 中用于容器管理的包，它基于 Go 语言实现，通过管理 namespaces、Cgroups、capabilities 以及文件系统来进行容器控制。runC 的 start 实际上是调用 libcontainer

的方法实现的。

runC 在 Github 的地址为 <https://github.com/opencontainers/runc>。

先从 main.go 看起, 在该文件中可以看到 runC 的基本命令只有常用的几个管理容器的命令, 包括启动、创建、停止、删除、恢复、迁移等。

```
// 代码地址: https://github.com/opencontainers/runc/blob/master/main.go#L93
app.Commands = []cli.Command{
    checkpointCommand,
    createCommand,
    deleteCommand,
    eventsCommand,
    execCommand,
    initCommand,
    killCommand,
    listCommand,
    pauseCommand,
    psCommand,
    restoreCommand,
    resumeCommand,
    runCommand,
    specCommand,
    startCommand,
    stateCommand,
    updateCommand,
}
```

因为 runC 核心部分还是 libcontainer, 所以 runC 实际上还是通过 libcontainer 的接口管理容器的。

而 libcontainer 的原理和 LXC 有些类似, 但是比 LXC 更强大、整合度更高。libcontainer 把容器技术的各种工具 (namespace、cgroup、rootfs 等) 整合抽象, 并向上层开放接口, 使得像如 Docker 等工具可以轻松与内核打交道, 如图 2.5 展示了 libcontainer 与其他技术在 Linux 内核通信过程的异同。

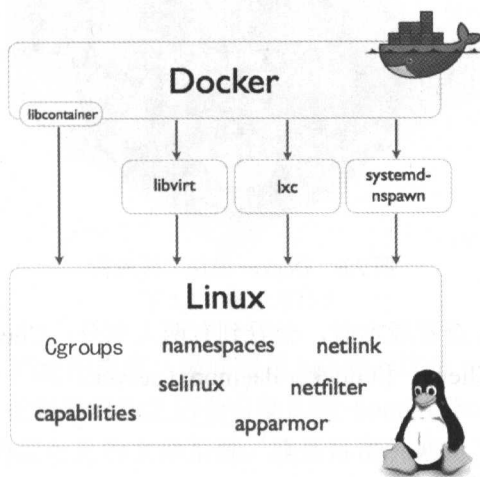


图 2.5 Docker 如何和 Linux 内核打交道

创建容器的时候, Docker 收集信息存到 config 中, 然后调用 libcontainer 的接口来实现容器的创建。runC 也一样, 不过把 Docker 自动收集信息这一步变成了手动操作, 用户

需要编写配置文件才能通过 libcontainer 启动一个容器（上文通过 `runc spec` 生成的是标准配置文件，事实上很多时候需要手动修改配置）。

2.5 Docker 基本架构

Docker 是一个构建、发布、运行分布式应用的平台，Docker 平台由 Docker Engine（运行环境 + 打包工具）、Docker Hub（API + 生态系统）两部分组成。Docker 的底层是各种 OS 以及云计算基础设施，而上层则是各种应用程序和管理工具，每层之间都是通过 API 来通信的。

2.5.1 Docker Client 介绍

如图 2.6 所示，Docker 引擎可以直观理解为就是在某一台机器上运行的 Docker 程序，实际上它是一个 C/S 结构的软件，有一个后台守护进程在运行，每次运行 `docker` 命令的时候实际上都是通过 RESTful Remote API 来和守护进程进行交互的，即使是在同一台机器上也是如此。

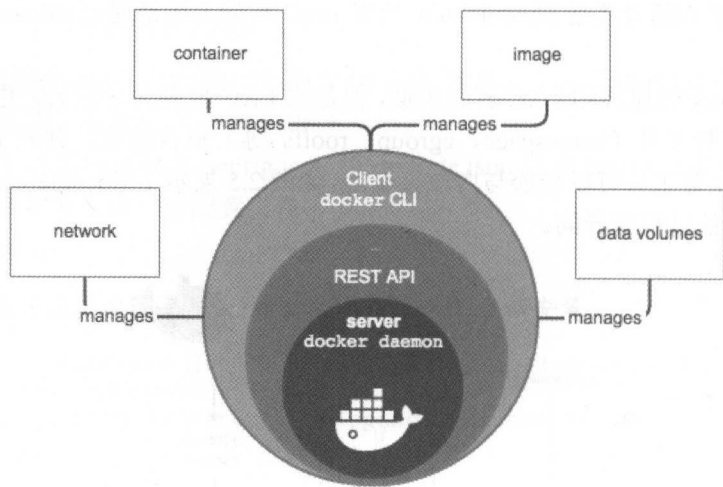


图 2.6 Docker 模块工作流程

在使用 `docker version` 查看版本时，会看到有两大部分：Client 和 Server，其实这就是图 2.6 中的 docker CLI（Client）和 docker daemon（server）。

```
$ docker version
Client:
Version:      1.12.1
API version:  1.24
Go version:   go1.6.3
Git commit:   23cf638
Built:        Thu Aug 18 05:02:53 2016
OS/Arch:     linux/amd64
Server:
```

```
Version:      1.12.1
API version:  1.24
Go version:   go1.6.3
Git commit:   23cf638
Built:        Thu Aug 18 05:02:53 2016
OS/Arch:      linux/amd64
```

2.5.2 Docker daemon 介绍

如前面所说，daemon 就是一个守护进程，它实际上就是驱动整个 Docker 的核心引擎，在 Docker0.9 版本之前的 Docker 客户端和服务端是统一在同一个二进制文件中的，后来为了更好地对 Docker 模块进行管理，划分为四个二进制文件，分别是 docker、containerd、docker-containerd-shim 和 docker-runc。

划分开之后，守护进程与容器管理不再互相牵制，这也使得 Docker 支持热更新，更人性化。

2.5.3 Docker 镜像

Docker 镜像是 Docker 系统中的构建模块（Build Component），是启动一个 Docker 容器的基础。

下面通过一个官方提供的示意图（如图 2.7 所示）来帮助理解镜像的概念。

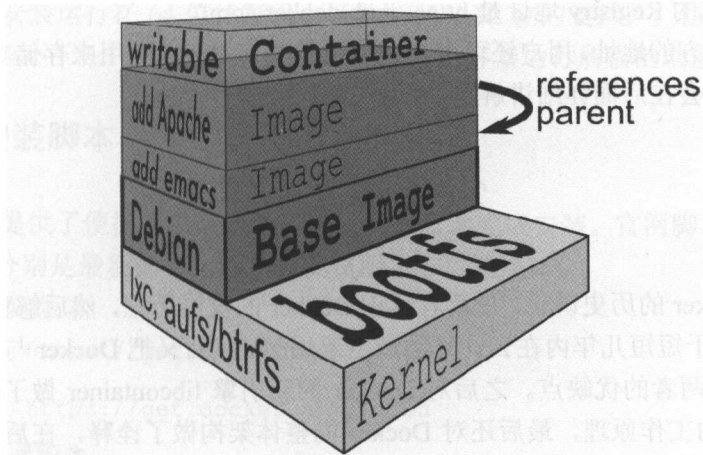


图 2.7 镜像是什么

Docker 镜像采用分层的结构构建，最底层是 bootfs，这是一个引导文件系统，一般用户很少会直接与其交互，在容器启动之后会自动卸载 bootfs，bootfs 之上是 rootfs，rootfs 是 Docker 容器在启动时内部可见的文件系统，就是日常所见的“/”目录。

Docker 镜像使用到了联合挂载技术和写时复制技术，关于这些内容会在第 5 章有详细介绍。利用这两项技术，Docker 可以只在文件系统发生变化时才会把文件写到可读写层，一层层叠加，这样不仅有利于版本管理，还利于存储管理。

2.5.4 Docker 容器

在 Docker 的世界中，容器是一个核心，容器是一个基于 Docker 镜像创建，包含了运行某一特定程序所有需要的 OS、软件、配置文件和数据，是一个可移植的运行单元。不过在宿主机看来，它只是一个简单的用户进程而已。关于容器的知识，在第 8 章会有详细介绍，在这里读者只需要知道容器是从镜像创建的运行实例，它是独立的一个沙盒即可。

容器很好地诠释了集装箱的理念，开发人员不用关心容器内部是什么应用，只管传输、运行即可，这是一种标准化的集装和运输方式，正是因为 Docker 把容器技术进行了体验友好的封装，才使得容器技术迅速推广普及。

2.5.5 Docker 仓库

相信大家对 Github 这个网站不会陌生，Github 上有着海量的代码仓库，类似的，在 Docker 中，当开发者想要构建一个镜像或运行一个容器时，一般要先有一个现成的镜像才可以执行构建或者运行，而本地又没有该特定镜像时怎么办呢？Docker 提出了 Registry 的概念，用户可以上传自己的镜像到 Registry 上，如果是公开的，那么全世界的用户都可以拉取这个镜像来操作，可以说 Registry 就是一个“软件商店”。类比传统运输业，Registry 类似一种船坞、中转站一样，是一个集中存放“集装箱”（镜像）的地方。

Docker 官方的 Registry 地址是 <https://hub.docker.com/>。

除了这个官方的地址，用户还可以搭建自己的私有 Registry 用来存储非公开的镜像等，关于这部分内容会在后面详细讲解。

2.6 本章小结

本章从 Docker 的历史讲起，使读者认识 Docker 的发展历程，然后解释 Docker 到底解决了什么，以至于短短几年内在云计算领域迅速崛起。然后又把 Docker 与虚拟机技术进行了比较，对比了两者的优缺点。之后对 Docker 的新引擎 libcontainer 做了剖析，使读者了解 libcontainer 的工作原理，最后还对 Docker 的整体架构做了诠释，在后续的章节中，将逐步对 Docker 的各个部分做详细讲解。

第3章 安 装 Docker

经过前面两章概念性极强的阅读，相信读者已经对 Docker 有了一定认识，如果读者看不懂，或者带有大量疑问也不用担心，因为前两章主要是以一个“目录”的形式给读者们一个了解 Docker 技术原理的途径。如果读者有兴趣可以深入了解，如果没兴趣也不影响后面对 Docker 的使用。

从本章开始，将进入有趣的实践部分了。截至 2016 年 10 月，Docker 已经原生支持 Linux、Windows、Mac OS 三大平台。本章将介绍在不同操作系统上安装 Docker 的方法。

3.1 Linux 系 统

Docker 基于 Linux 容器技术，面向服务器端，所以对 Linux 的支持无疑是最好的，主流的 Linux 系统都可以安装 Docker。

Docker 只能安装运行在 64 位计算机上（社区有对 32 位的支持），Linux 内核版本必须大于 3.10，内核小于 3.10 的系统会因为缺少 Docker 容器运行所需的功能而有错误。

3.1.1 一键安装脚本

Docker 官方提供了便捷的安装脚本，通过脚本自动完成安装。官网脚本一共有 3 种版本供用户选择，分别是最新的稳定版本、测试版本和实验版本。

（1）安装稳定版本

```
$ curl -sSL https://get.docker.com/ | sh
// 或者
$ wget -qO- https://get.docker.com/ | sh
```

（2）安装测试版本

```
$ curl -fsSL https://test.docker.com/ | sh
// 或者
$ wget -qO- https://test.docker.com/ | sh
```

（3）安装实验版本

```
$ curl -fsSL https://experimental.docker.com/ | sh
// 或者
$ wget -qO- https://experimental.docker.com/ | sh
```

如果脚本在用户的 Linux 机器上无法运行，或者出现其他错误，可以继续往下看，下文有非脚本的手动安装教程。

如果安装时出现如下没有 aufs 的提示，用户可以安装内核扩展（Ubuntu 系列）。

```
$ sudo apt-get install linux-image-extra-'uname -r'
```

或者自己下载 aufs 编译安装,又或者等待 10 秒,安装脚本会使用替代方案安装 Docker。

```
$ sudo curl -sSL https://get.docker.com/ | sh
modprobe: FATAL: Module aufs not found in directory /lib/modules/
4.4.0-2-**-amd64
Warning: current kernel is not supported by the linux-image-extra-virtual
package. We have no AUFS support. Consider installing the packages
linux-image-virtual kernel and linux-image-extra-virtual for AUFS support.
+ sleep 10
```

具体发行版安装会有细小差别,如果不能完成自动安装可以使用下面方法手动安装 Docker。

3.1.2 Debian 发行版

如前面所说,安装前确保机器为 64 位计算机,并且 Linux 内核在 3.10 以上。

1. 查看内核版本

要查看当前的内核版本,只需要打开一个终端,使用 `uname-r` 查看内核版本:

```
$ uname -r
3.13.0-85-generic
```

如果内核版本不达到要求,需要升级内核。目前 Debian 一般都不用升级内核。

```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
$ sudo reboot
```

2. 更新APT源

打开一个终端,安装 `apt-transport-https` 包,使得 APT 支持 HTTPS 协议的源。

```
$ sudo apt-get update && sudo apt-get install apt-transport-https
ca-certificates
```

添加 Docker 源的 gpg 密钥。

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADB76221572C52609D
```

3. 添加Docker的官方APT软件源

Debian 每次发行都有一个代号,使用下面命令可查看当前操作系统的代号。这里以 Debian 8.0 为例,返回的系统代号是 wheezy。

```
$ lsb_release -c
Codename: jessie
```

- Debian testing 代号为 stretch/Sid。
- Debian 8.0 代号为 Jessie。
- Debian 7.7 代号为 Wheezy。

获得系统代号之后，通过命令创建/etc/apt/sources.list.d/docker.list 文件（如果不存在就新建），并写入源的地址内容。注意修改为自己操作系统对应的代号。

创建/etc/apt/sources.list.d/docker.list 文件（这里使用编辑器 Vim，如果不熟悉 Vim 的操作，读者可以选择自己喜欢的编辑器进行编辑）。

```
$ sudo vim /etc/apt/sources.list.d/docker.list
```

Debian Wheezy 如下：

```
deb https://apt.dockerproject.org/repo debian-wheezy main
```

Debian Jessie 如下：

```
deb https://apt.dockerproject.org/repo debian-jessie main
```

Debian Stretch/Sid 如下：

```
deb https://apt.dockerproject.org/repo debian-stretch main
```

添加成功后，更新 APT 软件包缓存。

```
$ sudo apt-get update
```

校验软件包缓存结果。

```
$ apt-cache policy docker-engine
```

4. 安装 Docker

安装 Docker 之前，如果用户以前装过 Docker，那么需要先完全卸载 Docker 再安装：

```
$ apt-get purge "lxc-docker*"
$ apt-get purge "docker.io*"
```

一切没问题之后，执行安装。

```
$ sudo apt-get install docker-engine
```

5. 启动 Docker

```
$ sudo service docker start
```

6. 确认 Docker 运行正常

```
$ sudo docker run --rm hello-world
```

返回 Hello World 表示运行正常。

7. 为非root用户授权

如果没有 Docker 用户组就建立一个 Docker 用户组（默认安装后自动创建）。

```
$ sudo groupadd docker
```

增加当前用户到 Docker 组，需要注销来生效。

```
$ sudo gpasswd -a${USER} docker
```

重启 Docker 服务。

```
$ sudo service docker restart
```

这样，执行 Docker 命令就不必使用 sudo 申请权限了。

3.1.3 Ubuntu 发行版

Docker 目前支持的最低 Ubuntu 版本为 12.04LTS, 但推荐用户至少使用 14.04LTS 版本。

1. 查看内核版本

要查看当前的内核版本, 只需要打开一个终端, 使用 `uname-r` 查看内核版本。

```
$ uname -r
3.13.0-85-generic
```

如果内核版本不达到要求, 需要升级内核。

```
$ sudo apt-get update
$ sudo apt-get install -y linux-images-generic-lts-raring linux-headers-
generic-lts-raring
$ sudo reboot
```

2. 更新APT源

打开一个终端, 安装 `apt-transport-https` 包, 使得 APT 支持 HTTPS 协议的源。

```
$ sudo apt-get update && sudo apt-get install apt-transport-https
ca-certificates
```

添加 Docker 源的 `gpg` 密钥。

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

3. 添加Docker的官方APT软件源

Ubuntu / Debian 每次发行都有一个代号, 使用下面命令可查看当前操作系统的代号。

这里以 Ubuntu 14.04 为例, 返回的系统代号是 `trusty`。

```
$ lsb_release -c
Codename: trusty
```

- Ubuntu 12.04 (LTS)代号为 `precise`。
- Ubuntu 14.04 (LTS)代号为 `trusty`。
- Ubuntu 15.04 代号为 `vivid`。
- Ubuntu 15.10 代号为 `wily`。
- Ubuntu 16.04 (LTS)代号为 `xenial`。

获得系统代号之后, 通过命令创建 `/etc/apt/sources.list.d/docker.list` 文件 (如果不存在就新建), 并写入源的地址内容。注意修改为自己操作系统对应的代号。

创建 `/etc/apt/sources.list.d/docker.list` 文件 (这里使用编辑器 Vim, 如果不熟悉 Vim 的操作, 可以选择自己喜欢编辑器进行编辑)。

```
$ sudo vim /etc/apt/sources.list.d/docker.list
```

Ubuntu 12.04 (LTS)如下:

```
deb https://apt.dockerproject.org/repo ubuntu-precise main
```

Ubuntu 14.04 (LTS 如下):

```
deb https://apt.dockerproject.org/repo ubuntu-trusty main
```

Ubuntu 15.04 如下:

```
deb https://apt.dockerproject.org/repo ubuntu-vivid main
```

Ubuntu 15.10 如下:

```
deb https://apt.dockerproject.org/repo ubuntu-wily main
```

Ubuntu 16.04 (LTS) 如下:

```
deb https://apt.dockerproject.org/repo ubuntu-xenial main
```

添加成功后,更新 APT 软件包缓存。

```
$ sudo apt-get update
```

校验软件包缓存结果:

```
$ apt-cache policy docker-engine
```

4. 安装 Docker

安装 Docker 之前,如果用户使用的是 Ubuntu 12.04 可以先升级系统。

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

```
$ sudo reboot
```

确保基本安装条件满足,如果用户以前装过 Docker,那么需要先完全卸载 Docker 再安装。

```
$ apt-get purge "lxc-docker*"
```

```
$ apt-get purge "docker.io*"
```

然后再执行如下安装步骤。

```
$ sudo apt-get update
```

```
$ sudo apt-get install docker-engine
```

正在读取软件包列表...

正在分析软件包的依赖关系树...

正在读取状态信息...

将会同时安装下列软件:

```
aufs-tools cgroupfs-mount
```

下列【新】软件包将被安装:

```
aufs-tools cgroupfs-mount docker-engine
```

升级了 0 个软件包,新安装了 3 个软件包,要卸载 0 个软件包,有 6 个软件包未被升级。

需要下载 14.6 MB 的归档。

解压缩后会消耗 73.7 MB 的额外空间。

```
获取:1 http://cn.archive.ubuntu.com/ubuntu xenial/universe amd64 aufs-  
tools amd64 1:3.2+20130722-1.1ubuntu1 [92.9 kB]
```

```
获取:2 http://cn.archive.ubuntu.com/ubuntu xenial/universe amd64 cgroupfs-  
mount all 1.2 [4,970 B]
```

```
获取:3 https://apt.dockerproject.org/repo ubuntu-xenial/main amd64 docker-  
engine amd64 1.11.2-0~xenial [14.5 MB]
```

已下载 14.6 MB,耗时 3 分 52 秒 (62.5 kB/s)

正在选中未选择的软件包 aufs-tools。

(正在读取数据库 ... 系统当前共安装有 195748 个文件和目录。)

```
正准备解包 .../aufs-tools_1%3a3.2+20130722-1.1ubuntu1_amd64.deb ...
```

```
正在解包 aufs-tools (1:3.2+20130722-1.1ubuntu1) ...
```

正在选中未选择的软件包 cgroupfs-mount。

```
正准备解包 .../cgroupfs-mount_1.2_all.deb ...
```

```
正在解包 cgroupfs-mount (1.2) ...
```

正在选中未选择的软件包 docker-engine。

```

正准备解包 .../docker-engine_1.11.2-0~xenial_amd64.deb ...
正在解包 docker-engine (1.11.2-0~xenial) ...
正在处理用于 libc-bin (2.23-0ubuntu3) 的触发器 ...
正在处理用于 man-db (2.7.5-1) 的触发器 ...
正在处理用于 ureadahead (0.100.0-19) 的触发器 ...
ureadahead will be reprofiled on next reboot
正在处理用于 systemd (229-4ubuntu6) 的触发器 ...
正在设置 aufs-tools (1:3.2+20130722-1.1ubuntu1) ...
正在设置 cgroupfs-mount (1.2) ...
正在设置 docker-engine (1.11.2-0~xenial) ...
正在处理用于 libc-bin (2.23-0ubuntu3) 的触发器 ...
正在处理用于 systemd (229-4ubuntu6) 的触发器 ...
正在处理用于 ureadahead (0.100.0-19) 的触发器 ...
+ sudo -E sh -c docker version

```

Client:

```

Version:      1.11.2
API version:  1.23
Go version:   go1.5.4
Git commit:   b9f10c9
Built:        Wed Jun 1 22:00:43 2016
OS/Arch:      linux/amd64

```

Server:

```

Version:      1.11.2
API version:  1.23
Go version:   go1.5.4
Git commit:   b9f10c9
Built:        Wed Jun 1 22:00:43 2016
OS/Arch:      linux/amd64

```

If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

```
sudo usermod -aG docker yourusername
```

Remember that you will have to log out and back in for this to take effect!

5. 启动Docker

```
$ sudo service docker start
```

6. 确认Docker运行正常

```
$ sudo docker run --rm hello-world
```

返回 Hello World 表示运行正常。

7. 为非root用户授权

如果没有 Docker 用户组就建立一个 Docker 用户组（默认安装后自动创建）。

```
$ sudo groupadd docker
```

增加当前用户到 Docker 组，需要注销来生效：

```
$ sudo gpasswd -a${USER} docker
```

重启 Docker 服务：

```
$ sudo service docker restart
```

这样，执行 Docker 命令就不必使用 sudo 申请权限了。

3.1.4 Centos/Fedora 发行版

Docker（重新编译自 RHEL7）已收录在 CentOS-Extras 软件库内。用户只需执行以下安装命令即可。

```
[root@VM_75_19_centos ~]# sudo yum install docker
已加载插件: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
正在解决依赖关系
--> 正在检查事务
---> 软件包 docker.x86_64.0.1.10.3-46.el7.centos.10 将被 安装
--> 正在处理依赖关系 docker-common = 1.10.3-46.el7.centos.10, 它被软件包
docker-1.10.3-46.el7.centos.10.x86_64 需要
--> 正在处理依赖关系 oci-systemd-hook >= 1:0.1.4-4, 它被软件包
docker-1.10.3-46.el7.centos.10.x86_64 需要
--> 正在处理依赖关系 oci-register-machine >= 1:0-1.7, 它被软件包
docker-1.10.3-46.el7.centos.10.x86_64 需要
--> 正在处理依赖关系 docker-selinux >= 1.10.3-46.el7.centos.10, 它被软件包
docker-1.10.3-46.el7.centos.10.x86_64 需要
--> 正在检查事务
---> 软件包 docker-common.x86_64.0.1.10.3-46.el7.centos.10 将被 安装
---> 软件包 docker-selinux.x86_64.0.1.10.3-46.el7.centos.10 将被 安装
---> 软件包 oci-register-machine.x86_64.1.0-1.7.git31bbcd2.el7 将被 安装
---> 软件包 oci-systemd-hook.x86_64.1.0.1.4-4.git41491a3.el7 将被 安装
--> 解决依赖关系完成
```

依赖关系解决

Package	架构	版本	源	大小
正在安装:				
docker	x86_64	1.10.3-46.el7.centos.10	extras	9.5 M
为依赖而安装:				
docker-common	x86_64	1.10.3-46.el7.centos.10	extras	61 k
docker-selinux	x86_64	1.10.3-46.el7.centos.10	extras	78 k
oci-register-machine	x86_64	1:0-1.7.git31bbcd2.el7	extras	929 k
oci-systemd-hook	x86_64	1:0.1.4-4.git41491a3.el7	extras	27 k

事务概要

```
=====
安装 1 软件包 (+4 依赖软件包)
总下载量: 11 M
安装大小: 48 M
Is this ok [y/d/N]: y
Downloading packages:
(1/5): docker-common-1.10.3-46.el7.centos.10.x86_64.rpm
| 61 kB 00:00:00
(2/5): docker-selinux-1.10.3-46.el7.centos.10.x86_64.rpm
| 78 kB 00:00:00
(3/5): oci-register-machine-0-1.7.git31bbcd2.el7.x86_64.rpm
```

```
| 929 kB 00:00:00
(4/5): oci-systemd-hook-0.1.4-4.git41491a3.el7.x86_64.rpm
| 27 kB 00:00:00
(5/5): docker-1.10.3-46.el7.centos.10.x86_64.rpm
| 9.5 MB 00:00:01
```

总计

7.9 MB/s | 11 MB 00:00:01

Running transaction check

Running transaction test

Transaction test succeeded

Running transaction

正在安装 : 1:oci-register-machine-0-1.7.git31bbcd2.el7.x86_64
1/5

正在安装 : docker-selinux-1.10.3-46.el7.centos.10.x86_64
2/5

正在安装 : 1:oci-systemd-hook-0.1.4-4.git41491a3.el7.x86_64
3/5

正在安装 : docker-common-1.10.3-46.el7.centos.10.x86_64
4/5

正在安装 : docker-1.10.3-46.el7.centos.10.x86_64
5/5

验证中 : docker-common-1.10.3-46.el7.centos.10.x86_64
1/5

验证中 : docker-1.10.3-46.el7.centos.10.x86_64
2/5

验证中 : 1:oci-systemd-hook-0.1.4-4.git41491a3.el7.x86_64
3/5

验证中 : docker-selinux-1.10.3-46.el7.centos.10.x86_64
4/5

验证中 : 1:oci-register-machine-0-1.7.git31bbcd2.el7.x86_64
5/5

已安装:

docker.x86_64 0:1.10.3-46.el7.centos.10

作为依赖被安装:

docker-common.x86_64 0:1.10.3-46.el7.centos.10

docker-selinux.x86_64 0:1.10.3-46.el7.centos.10

oci-register-machine.x86_64 1:0-1.7.git31bbcd2.el7

oci-systemd-hook.x86_64 1:0.1.4-4.git41491a3.el7

完毕!

[root@VM_75_19_centos ~]#

如果用户想采用一个较新版本的 Docker，则有两个选择：

(1) 使用来自 Fedora 的组件。

```
$ sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
```

```
[virt7-docker-fedora-candidate]
```

```
name=virt7-docker-fedora-candidate
```

```
baseurl=http://cbs.centos.org/repos/virt7-docker-fedora-candidate/x86_64/os/
```

```
enabled=1
```

```
gpgcheck=0
```

```
EOF
```

(2) 使用来自 RHEL 的组件。

```
$ sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
[virt7-docker-el-candidate]
name=virt7-docker-el-candidate
baseurl=http://cbs.centos.org/repos/virt7-docker-el-candidate/x86_64/os/
enabled=1
gpgcheck=0
EOF
```

⚠注意：在系统上同时启用这两个软件库会混淆来自不同源头的组件而导致无法预知的后果。同时或许需要停用 CentOS-Extras，以确保安装的组件是来自虚拟化 SIG 软件库。

```
$ sudo yum install docker --disablerepo=extras
```

安装 Docker 后，必须引导该服务才能应用它。

```
$ sudo systemctl start docker
```

若要开机时引导 Docker 服务：

```
$ sudo systemctl enable docker
```

或者：

```
$ sudo chkconfig docker on
```

可使用如下命令查看一下使用仓库安装的 Docker 信息。

```
[root@VM_75_19_centos ~]# docker --version
Docker version 1.10.3, build d381c64-unsupported
```

下面尝试使用官方的脚本安装，CentOS 7.1 安装 Docker。

```
[root@VM_75_19_centos ~]# curl -sSL https://get.docker.com/ | sh
+ sh -c 'sleep 3; yum -y -q install docker-engine'
warning:
/var/cache/yum/x86_64/7/docker-main-repo/packages/docker-engine-selinux
-1.12.1-1.el7.centos.noarch.rpm: Header V4 RSA/SHA512 Signature, key ID
2c52609d: NOKEY
```

docker-engine-selinux-1.12.1-1.el7.centos.noarch.rpm 的公钥尚未安装
导入 GPG key 0x2C52609d:

```
用户 ID      : "Docker Release Tool (release docker) <docker@docker.com>"
指纹        : 5811 8e89 f3a9 1289 7c07 0adb f762 2157 2c52 609d
来自        : https://yum.dockerproject.org/gpg
```

setsebool: SELinux is disabled.

If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

```
sudo usermod -aG docker your-user
```

Remember that you will have to log out and back in for this to take effect!

```
[root@VM_75_19_centos ~]#
```

查看安装的 Docker 信息如下：

```
[root@VM_75_19_centos ~]# docker info
Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
Server Version: 1.12.1
```

```
Storage Driver: devicemapper
Pool Name: docker-253:1-32927-pool
Pool Blocksize: 65.54 kB
Base Device Size: 10.74 GB
Backing Filesystem: xfs
Data file: /dev/loop0
Metadata file: /dev/loop1
Data Space Used: 11.8 MB
Data Space Total: 107.4 GB
Data Space Available: 19.54 GB
Metadata Space Used: 581.6 kB
Metadata Space Total: 2.147 GB
Metadata Space Available: 2.147 GB
Thin Pool Minimum Free Space: 10.74 GB
Udev Sync Supported: true
Deferred Removal Enabled: false
Deferred Deletion Enabled: false
Deferred Deleted Device Count: 0
Data loop file: /var/lib/docker/devicemapper/devicemapper/data
WARNING: Usage of loopback devices is strongly discouraged for production
use. Use '--storage-opt dm.thinpooldev' to specify a custom block storage
device.
Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
Library Version: 1.02.107-RHEL7 (2015-10-14)
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge null host overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: seccomp
Kernel Version: 3.10.0-327.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 993.1 MiB
Name: VM_75_19_centos
ID: A7RO:EEJV:V4K7:464M:PMPW:LLQA:CWCJ:L3Q6:V4OR:736I:ZQS2:7S3Z
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
Insecure Registries:
  127.0.0.0/8
[root@VM_75_19_centos ~]#
```

可以看到，从官方脚本安装的 Docker 比仓库里的要新，读者可以按情况安装。

在 CentOS 6.5 上安装 Docker 需要采用 EPEL 软件库。启用 EPEL 后，才能继续以下的安装程序。

要在 CentOS 6 上安装 Docker，可利用以下指令安装 docker-io 组件。

```
$ sudo yum install docker-io
```

如果不能直接安装 **docker-io**，可利用 EPEL 软件库。

```
$ rpm -i Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
$ yum update -y
```

安装 Docker 后，必须引导该服务才能应用它。

```
$ sudo service docker start
```

若要开机时引导 Docker 服务：

```
$ sudo chkconfig docker on
```

查看版本信息：

```
[root@VM_75_19_centos ~]# docker --version
Docker version 1.7.1, build 786b29d/1.7.1
```

使用官方脚本安装：

```
[root@VM_75_19_centos ~]# curl -sSL https://get.docker.com/ | sh
+ sh -c 'sleep 3; yum -y -q install docker-engine'
If you would like to use Docker as a non-root user, you should now consider
adding your user to the "docker" group with something like:
    sudo usermod -aG docker your-user
Remember that you will have to log out and back in for this to take effect!
[root@VM_75_19_centos ~]#
```

查看版本：

```
[root@VM_75_19_centos ~]# docker --version
Docker version 1.7.1, build 786b29d
```

目前 CentOS 6.5 的 Linux 内核版本是 2.6.32：

```
[root@VM_75_19_centos ~]# uname -a
Linux VM_75_19_centos 2.6.32-504.30.3.el6.x86_64 #1 SMP Wed Jul 15 10:13:09
UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
```

虽然可以安装，但是并不能使用 Docker 的全部功能。关于非 root 使用 Docker 的方法和前面一样，不再赘述。

3.1.5 Arch Linux 发行版

使用 Arch Linux 的用户不在少数，可以使用 Arch Linux 社区发布的 Docker 软件包进行安装，名字为 **docker**，或者使用 AUR 包，名字为 **docker-git**。

docker 软件包将会安装最新正式版本的 Docker。

docker-git 则是由当前 master 分支构建的包，属于 test 版本。

Docker 依赖于几个指定的安装包，核心的几个依赖包为 **bridge-utils**、**device-mapper**、**iproute2**、**sqlite**。所以社区包安装很简单：

```
$ sudo pacman -S docker
```

这样就安装了你所需要的一切。

如果想使用 AUR 的包，则执行：

```
$ sudo yaourt -S docker-git
```

这里假设用户已经安装好了 **yaourt**，如果之前没有安装构建过这个包，请参考 Arch User Repository。

启动 Docker。

```
$ sudo systemctl start docker
```

设置开机启动：

```
$ sudo systemctl enable docker
```

如果执行 `Docker info` 可以返回正确信息，那么 Arch Linux 已经成功安装并运行 Docker 了。

3.1.6 Suse/openSUSE 发行版

Docker 支持 openSUSE 12.3 或更高版本。由于 Docker 的限制，Docker 只能运行在 64 位的主机上。

Docker 不被包含在 openSUSE 12.3 和 openSUSE 13.1 的官方镜像仓库中，因此需要添加 OBS 的虚拟化仓库来安装 Docker 包。

可执行下面的命令来添加虚拟化仓库。

openSUSE 12.3 如下：

```
$ sudo zypper ar -f \
    http://download.opensuse.org/repositories/Virtualization/openSUSE_12.3/ \
    Virtualization
```

openSUSE 13.1 如下：

```
$ sudo zypper ar -f \
    http://download.opensuse.org/repositories/Virtualization/openSUSE_13.1/ \
    Virtualization
```

在 openSUSE 13.2 版本以后就不需要添加额外的库了。

安装 Docker 包。

```
$ sudo zypper in docker
```

启动 Docker 进程。

```
$ sudo systemctl start docker
```

设置开机启动 Docker。

```
$ sudo systemctl enable docker
```

如果执行 `Docker info` 可以返回正确信息，那么 openSUSE 已经成功安装并运行 Docker 了。

3.2 Windows 与 Mac OS 系统

在 Docker 支持 Windows 平台之后，用户可以有两种方式安装 Docker，一是传统的安装 Linux 虚拟机，然后在虚拟机里安装 Docker；另一种是直接安装 Docker for Windows。显然后一种原生应用更吸引人，但是 Docker for Windows 只支持 Windows 10 的 64 位专业版（企业版）或者 Windows Server 2016。

本书不讲述使用 Docker Toolbox 安装 Docker 的方法，因为 Docker Toolbox 本质就是虚拟机，Docker Toolbox 先在虚拟机里安装 Linux，然后再在 Linux 上安装 Docker。关于 Docker Toolbox 的文档，读者可以在 <https://docs.docker.com/toolbox/overview/> 上找到。

Mac OS 版的 Docker 出现比 Windows 版早，目前 Docker for Mac OS 要求 Mac OS 版本大于 10.10.3 Yosemite。因为用到新处理器的特性，还要求是 2010 年之后的 Mac 机器。

3.2.1 在 Windows 上安装原生 Docker

如果是 Windows 10 的 64 位专业版或企业版用户，那么可以直接安装原生 Docker 应用。

先从官网下载安装文件，打开 <https://docker.com> 之后往下拉就可以看到下载链接（如图 3.1 所示），其中有 Windows 和 Mac OS 版本，这里是稳定版的 Docker，如果想体验测试版，可以选择 Beta 下载。

稳定版下载网址为 <https://download.docker.com/win/stable/InstallDocker.msi>。

测试版下载网址为 <https://download.docker.com/win/beta/InstallDocker.msi>。

下载之后双击打开，界面如图 3.2 所示，选择同意就可以安装了。

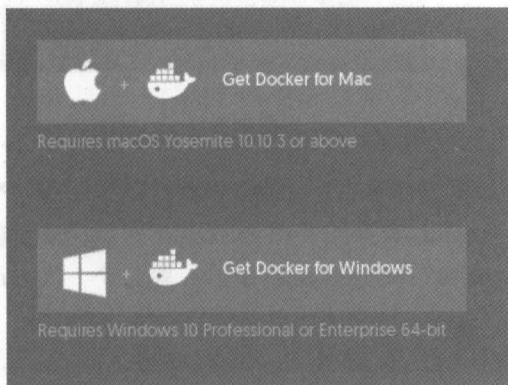


图 3.1 下载链接

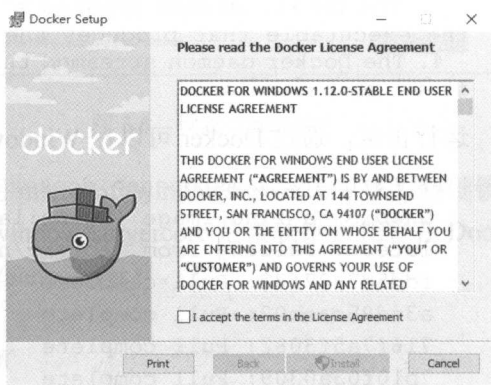


图 3.2 安装界面

安装之后，打开 Docker，可以看到状态栏有一个鲸鱼图标，表示 Docker 已经在运行了。如图 3.3 所示为打开 Docker 之后的界面。

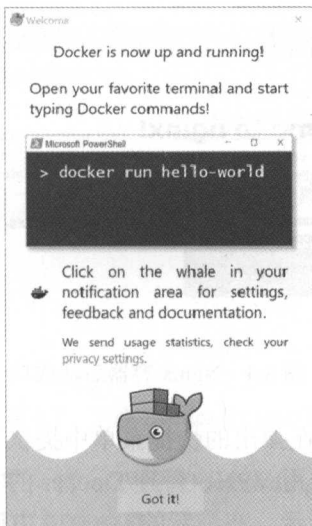


图 3.3 Docker 界面

选择 About Docker 可以查看 Docker 的版本,也可以在 cmd 中查看。

```
PS C:\Users\samstevens> docker --version
Docker version 1.12.0, build 8eab29e, experimental
PS C:\Users\samstevens> docker-compose --version
docker-compose version 1.8.0, build d988a55
PS C:\Users\samstevens> docker-machine --version
docker-machine version 0.8.0, build b85aac1
```

在 Windows 下使用 Docker 和 Linux 完全一样,这得益于前面提到的 libcontainer 的封装抽象。下面通过一个 Hello-World 来验证 Docker 是否能正常运行。

```
PS C:\Users\samstevens> docker pull hello-world
PS C:\Users\samstevens> docker run hello-world
Hello from Docker.
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs
the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent
it to your terminal.
```

运行正常,通过 Docker 可以在 Windows 上快速搭建一个基于 Linux 的 Nginx 环境。

```
PS C:\Users\samstevens> docker run -d -p 80:80 --name webserver nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fdd5d7827f33: Pull complete
a3ed95caeb02: Pull complete
716f7a5f3082: Pull complete
7b10f03a0309: Pull complete
Digest: sha256:f6a001272d5d324c4c9f3f183e1b69e9e0ff12debeb7a092730d638
c33e0de3e
Status: Downloaded newer image for nginx:latest
dfe13c68b3b86f01951af617df02be4897184cbf7a8b4d5caf1c3c5bd3fc267f
```

打开浏览器,输入 <http://localhost> 可以看到 Nginx 已经成功运行,如图 3.4 所示。

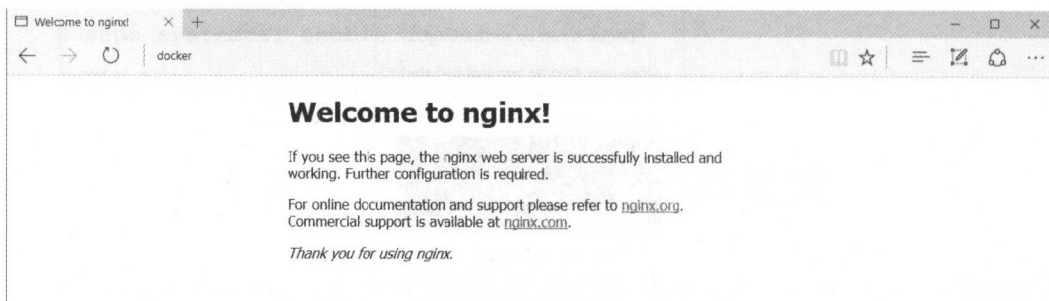


图 3.4 Nginx 容器启动成功

在 Windows 下右击图标,在弹出的快捷菜单中提供了自动启动、自动检查更新、与 Docker 容器共享本地驱动器、使用 VPN、管理 Docker 的 CPU 和内存使用、重启 Docker、重置 Docker 等功能,如图 3.5 所示。在最新的 Beta 26 中还提供了 Windows 和 Linux 容器之间切换。

如图 3.6 所示，还可以在下面设置中设置仓库镜像地址，避免因为网络问题无法拉取镜像，详细设置方法见第 7 章。

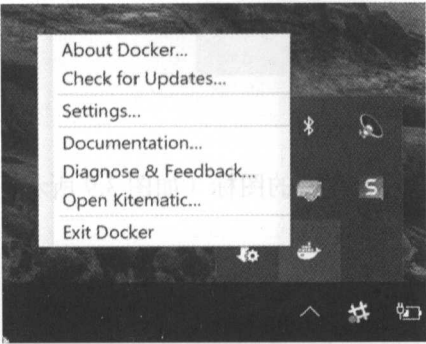


图 3.5 右键快捷菜单

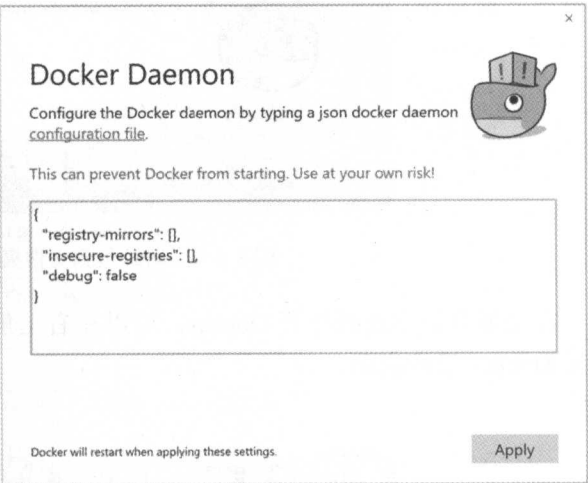


图 3.6 设置 Daemon

除了使用常规的镜像，用户目前还可以使用 microsoft/windowsservercore 作为基础镜像构建专门运行 Windows 服务的镜像，microsoft/windowsservercore 是微软维护托管在 Docker Hub 的一个镜像（包括 Nano Server: microsoft/nanoserver）。

3.2.2 在 Mac OS 上安装原生 Docker

Mac OS 的下载链接的网址为 <https://docs.docker.com/docker-for-mac/>，如图 3.7 所示，可以选择稳定版（Stable）或者是测试版（Beta）。

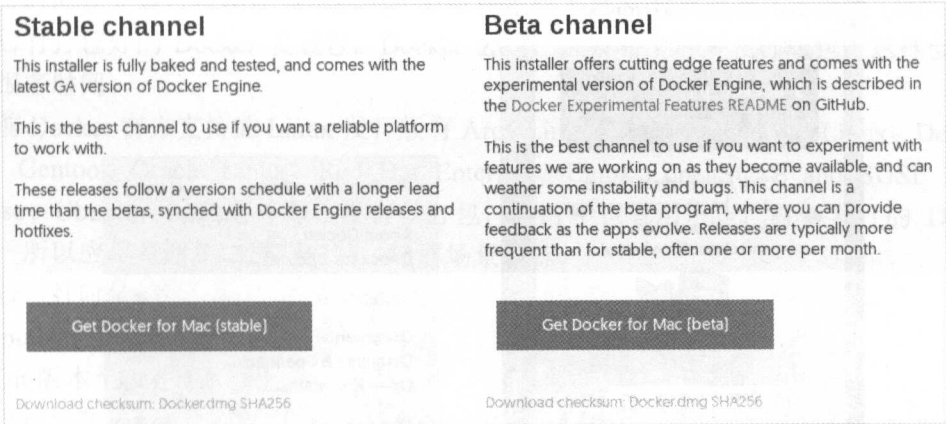


图 3.7 下载 Docker for macOS

安装方法和普通的 dmg 程序一样，双击 Docker.dmg 就可以打开安装程序，如图 3.8 所示，拖动鲸鱼到应用文件夹即可完成安装。

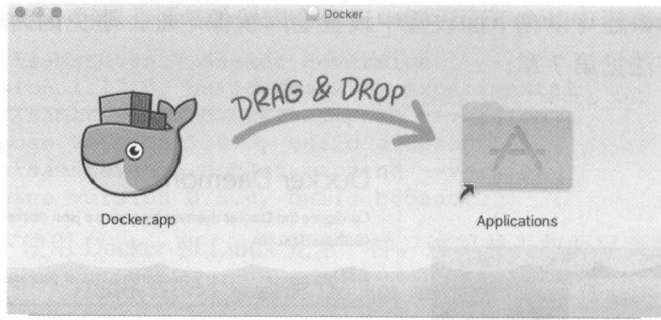


图 3.8 拖动鲸鱼到文件夹即可完成安装

在全屏菜单中选择打开 Docker，可以在右上角看到 Docker 的图标（如图 3.9 所示），表示 Docker 已经启动。

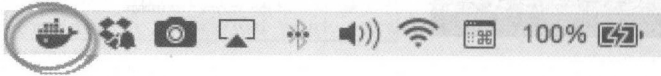


图 3.9 Docker 已经启动

单击图标，可以看到 Docker 的界面（如图 3.10 所示），与 Linux 一样，Docker 通过命令行来操作。例如查看 Docker 版本的命令如下：

```
$ docker --version
Docker version 1.12.0, build 8eab29e
```

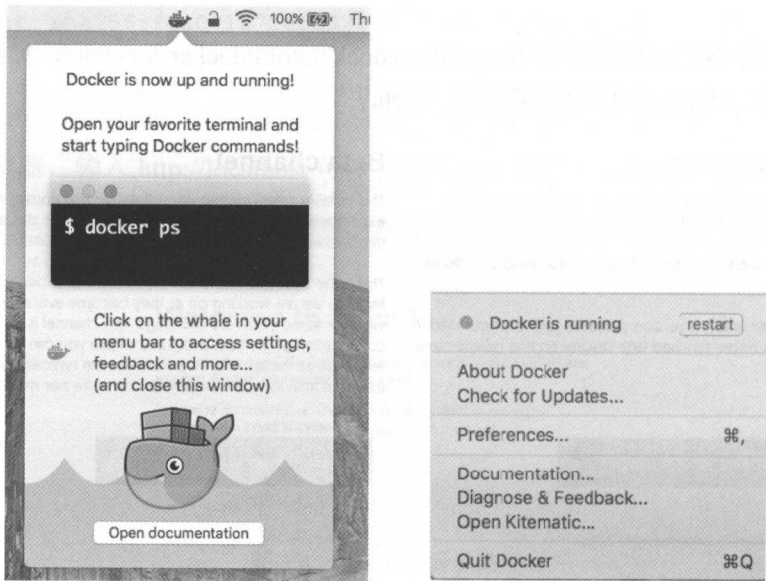


图 3.10 Docker 界面

同样可以通过运行一个 Nginx 命令来测试 Docker 是否运行正常：

```
$ docker run -d -p 80:80 --name webserver nginx
```

在浏览器中打开 `http://localhost/`，如果能看到 Nginx 的欢迎界面，说明 Docker 运行正常了。

Docker for Mac OS 的设置和 Windows 基本一致，都是可视化的操作，可以做一些基本的调整，如图 3.11 所示。

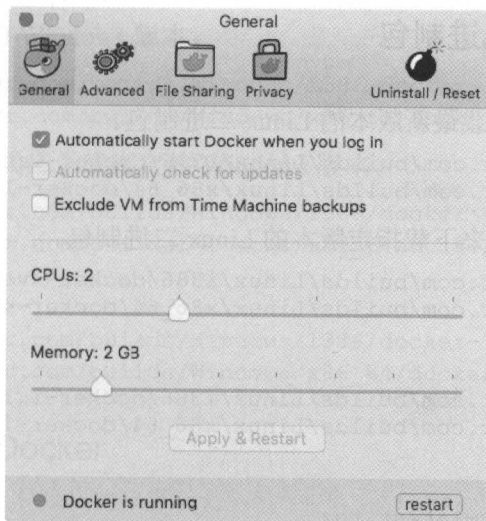


图 3.11 Docker for Mac OS 设置界面

3.3 二进制安装

一般情况下不推荐二进制安装方式，一是因为这个方法在安装之后可能会有诸多问题，二是浪费时间在安装过程实在得不偿失。所以在进行二进制安装之前，先检查 Linux 发行版本是否有打包好的 Docker 安装包。Docker 公司已经发布了许多发行版包，这样会节省用户的很多时间。

目前 Docker 官方支持的 Linux 发行版有 Arch Linux、CentOS、CRUX Linux、Debian、Fedora、Gentoo、Oracle Linux、Red Hat Enterprise Linux、openSUSE and SUSE Linux Enterprise、Ubuntu，除此之外即使没有官方包，社区中也有用户自己打包发行的 Docker 软件包，所以应尽量避免二进制安装，除非场景特殊。

检查运行时依赖：

- iptables 版本至少在 1.4 以上。
- Git 版本至少在 1.7 以上。
- 已经安装 Procps 或者类似能够提供 Ps 功能的软件。
- XZ Utils 版本在 4.9 以上。
- Linux kernel 至少 3.10 以上。

如果用户的 Linux 发行版上支持 AppArmor 或者 Selinux 请启用（大部分发行版默认情况下是启用 AppArmor 或者 Selinux），这样有助于提高安全性并阻止某些漏洞。关于如何

启动设置推荐的安全机制，在发行版提供的文档上提供了详细的步骤。

如果没问题的话，就可以从 Github 上获取稳定的发行版本号列表，在 [docker/docker](https://github.com/docker/docker) 发布页面可以获取到最新版的二进制文件。此外还可以通过官方地址 <https://github.com/docker/docker/releases> 获取该文件。

3.3.1 获取 Linux 二进制包

通过下面的链接来下载最新版本的 Linux 二进制包。

```
https://get.docker.com/builds/Linux/i386/docker-latest.tgz
https://get.docker.com/builds/Linux/x86_64/docker-latest.tgz
```

使用下面的链接模式来下载指定版本的 Linux 二进制包。

```
https://get.docker.com/builds/Linux/i386/docker-<version>.tgz
https://get.docker.com/builds/Linux/x86_64/docker-<version>.tgz
```

例如：

```
https://get.docker.com/builds/Linux/i386/docker-1.11.0.tgz
https://get.docker.com/builds/Linux/x86_64/docker-1.11.0.tgz
```

解压如下：

```
$ tar -xvzf docker-latest.tgz
docker/
docker/docker
docker/docker-containerd
docker/docker-containerd-ctr
docker/docker-containerd-shim
docker/docker-proxy
docker/docker-runc
docker/dockerd
```

安装运行：

```
$ chmod atx docker/*
$ mv docker/* /usr/bin/
$ sudo dockerd &
```

这样 Docker 就已经在后台运行了。

3.3.2 获取 Mac OS X 二进制包

Mac OS X 的二进制文件仅仅是一个客户端，不可以使用它来启动 Docker 进程。可以通过下面的链接来下载最新的 Mac OS X 版本：

```
https://get.docker.com/builds/Darwin/x86_64/docker-latest.tgz
```

通过下面的 URL 模式来下载指定的 Mac OS X 版本：

```
https://get.docker.com/builds/Darwin/x86_64/docker-<version>.tgz
```

例如：

```
https://get.docker.com/builds/Darwin/x86_64/docker-1.11.0.tgz
```

使用方法和 3.4.1 节的 Linux 类似，不再赘述。

3.3.3 获取 Windows 的二进制包

用户只能下载 1.9.1 之后的 Windows 二进制版本。此外，32 位二进制文件只有一个客户端，不可以使用它来启动 Docker 进程。64 位二进制包含客户端和守护进程。可以通过下面的链接来下载最新的 Windows 版本。

```
https://get.docker.com/builds/Windows/i386/docker-latest.zip
https://get.docker.com/builds/Windows/x86_64/docker-latest.zip
```

通过下面的 URL 模式来下载指定的 Windows 版本。

```
https://get.docker.com/builds/Windows/i386/docker-<version>.zip
https://get.docker.com/builds/Windows/x86_64/docker-<version>.zip
```

例如：

```
https://get.docker.com/builds/Windows/i386/docker-1.11.0.zip
https://get.docker.com/builds/Windows/x86_64/docker-1.11.0.zip
```




3.3.4 树莓派安装 Docker

二进制 Docker 的一个典型应用场景就是树莓派了，不过即便是树莓派也已经有开发者打包了一个开箱即用的镜像。

为了简化安装步骤，可以直接使用 HypriotOS，这是一个已经预安装了 Docker Engine 1.11.1 的系统，基于 Debian 开发，下载地址为 <https://github.com/hypriot/image-builder-rpi/releases/>。

Mac OS 下用 flash 刻录，Linux 下用 dd 命令刻录，Windows 下用常见的刻录软件刻录到 SD 卡即可。安装步骤和传统树莓派一样，网上资料很多，这里不再赘述。

想要构建树莓派镜像，也有一个比较好的基础镜像推荐：resin/rpi-raspbian，下面第三个二维码就是该镜像的地址，可以像 FROM ubuntu:trusty 一样以 FROMresin/rpi-raspbian 做基础镜像。

		
Building Docker 1.12 on a Raspberry Pi	How to get Docker working on your favourite ARM board with HypriotOS	resin/rpi-raspbian

3.4 本章小结

本章是在最新版本的 Docker 1.12 发布之后写的,选取了最流行的几款 Linux 系统做例子,并且注明了安装过程中会遇到的问题。本章还补充了 Docker for Windows 和 Docker for macOS 的安装教程,虽然安装过程不是什么难事,但是如果不幸遇到了问题,最好的解决办法就是去官方文档寻求支持,或者通过搜索引擎寻求帮助。

本章最后还介绍了安装二进制的 Docker 版本,以及在树莓派上运行 Docker,经过笔者验证,树莓派上运行 Docker 还是差强人意的,虽然没有大问题,但是在镜像选择上最好自己使用 resin/rpi-raspbian 重新构建镜像。

本章结束后,将进入 Docker 的基础学习阶段,虽然难度不大,但是内容还是不少的,那么,前进吧。

第2篇

Docker 基础知识

» 第4章 Docker 基础

» 第5章 Docker 镜像

» 第6章 Dockerfile 文件

» 第7章 Docker 仓库

» 第8章 Docker 容器

» 第9章 数据卷

» 第10章 网络管理

第4章 Docker 基础

经过第3章的实践，相信读者已经成功在自己的计算机上安装部署了 Docker，本章开始带领读者初步熟悉 Docker 的基本操作。本章的目的是方便读者在后续学习时遇到 Docker 命令使用问题时，可以回到本章快速查阅相应操作。

本章主要有两大部分，第一部分是对 Docker 目前 45 个子命令的解释，第二部分是使用 Docker 学习最基本的两个操作——运行容器和构建镜像。

4.1 Docker 基本操作

经过第3章的实践，相信读者已经安装好了 Docker，从本章开始将是 Docker 的实践应用阶段。截至 Docker 1.12 版，Docker 一共有 45 个一级子命令，即如 `docker [command]` 这样的子命令。在本章节中并不会详细介绍全部的 45 个子命令，相对独立的命令，如 `login`、`version`、`info` 这些子命令，会尽量详细介绍，因为后面章节遇见的机会不多，而像 `run`、`build`、`network` 这些围绕镜像、容器、网络的命令，将会在后续章节占有较大篇幅，因此在本章中不作详细介绍（本书以 Docker 1.12 Stable 版本为准，在已知的 Experimental 版本中新添加的 3 个新的命令，即 `deploy`、`plugin`、`stack` 不在本书讨论范围内）。

在介绍 Docker 操作之前，先一个问题解决了，有使用 Docker 经验的读者可能会发现，非 root 用户在安装完 Docker 之后，如果直接使用 Docker 命令会出现如下信息：

```
$ docker ****
Cannot connect to the Docker daemon. Is the Docker daemon running on this host?
```

或者

```
Are you trying to connect to a TLS-enabled daemon without TLS?
```

之类的信息。导致这种情况一般有两种原因，一是 Docker 没有启动，在第3章中已经讲过如何启动 Docker 服务；另一种情况就是没有使用 `sudo docker` 的方式输入 Docker 命令。

原因：因为 `/var/run/docker.sock` 所属 Docker 组具有 `setuid` 权限，非 root 用户使用 `setuid` 这个权限需要申请。

对于后者，显然每次使用 Docker 都要使用 `sudo` 是一件很麻烦的事，所以先把 `sudo` 去掉，以便更好地操作。

一般安装 Docker 之后系统会自动创建一个 Docker 的用户组，如果没有创建可以手动创建。

```
$ sudo groupadd docker
```

将当前非 root 用户加入该 group 内，然后退出并重新登录就生效了。

```
$ sudo gpasswd -a${USER} docker
```

重启 Docker 服务。

```
$ sudo service docker restart
```

切换当前会话到新 group 或者重启 X 会话。

```
$ newgrp - docker
// or
$ pkill X
```

注意，最后一步是必须的，否则因为 groups 命令获取到的是缓存的组信息，刚添加的组信息未能生效，所以 Docker images 执行时同样有错。

然后打开终端，输入 docker 后按 Enter 键，可以看到输出的关于 Docker 的使用基本说明。

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]
       docker [ --help | -v | --version ]
```

A self-sufficient runtime for containers.

Options:

--config=~/ .docker	Location of client config files
-D, --debug	Enable debug mode
-H, --host=[]	Daemon socket(s) to connect to
-h, --help	Print usage
-l, --log-level=info	Set the logging level
--tls	Use TLS; implied by --tlsverify
--tlscacert=~/ .docker/ca.pem	Trust certs signed only by this CA
--tlscert=~/ .docker/cert.pem	Path to TLS certificate file
--tlskey=~/ .docker/key.pem	Path to TLS key file
--tlsverify	Use TLS and verify the remote
-v, --version	Print version information and quit

Commands:

```
.....
```

// 这里 45 个子命令将在下面介绍

Run 'docker COMMAND --help' for more information on a command.

Docker 命令分为管理命令、镜像命令、容器命令、仓库命令、网络命令、数据卷命令、编排命令等，命令的分类在每次版本发布时都有调整，所以这里列出的是 Docker 1.12.1 的命令分类。

本章信息很多，接下来将按照字母顺序一个个介绍，读者可以选择跳过部分内容，本章主要是为了以后使用过程中遇到不知如何操作时回来阅读用。

4.1.1 依附容器的 docker attach 命令

attach 的中文意思有附加、贴上、系上等意思，所以 docker attach 主要的作用就是进入容器，这个进入容器和后面的 docker exec 类似但是不完全一样。

说到进入容器，不论是开发者还是运维人员，都经常有这样的诉求。目前主要的方法不外乎以下 3 种：

- 使用 ssh 登录进容器。
- 使用 nsenter、nsinit 等第三方工具。
- 使用 Docker 本身提供的工具。

第1种方法需要在容器中启动 sshd, 违反了 Docker 所倡导的一个容器一个进程的原则。同时也存在开销和攻击面增大的问题。第2种方法需要额外学习使用第三方工具。

所以最好的方法当然是使用官方的工具, 下面来看一下首先登场的 attach 命令。

```
$ docker attach --help
```

```
Usage: docker attach [OPTIONS] CONTAINER
```

```
Attach to a running container
```

```
Options:
```

```
--detach-keys string    Override the key sequence for detaching a
container
--help                  Print usage
--no-stdin              Do not attach STDIN
--sig-proxy             Proxy all received signals to the process (default
true)
```

从上面可以看出, attach 的功能不多, 主要作用就是进入容器内部, 可以查看容器内部的持续输出, 或以交互方式控制容器, 下面以运行一个 Ubuntu 镜像为例进行说明。

```
$ docker run -itd --name ubuntu ubuntu:14.04
002b10021bc3f840eca4aa71194ce57962402217913b106be760cffd78829534
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS      NAMES
002b10021bc3   ubuntu:14.04 "/bin/bash"             17 seconds ago Up 14 seconds
ubuntu
$ docker attach ubuntu
root@002b10021bc3:/#
```

可以看到终端状态已经改变, 此时终端已经进入了容器内部, 现在可以执行一些命令, 如查看容器内部的网络情况。

```
root@002b10021bc3:/home# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:42 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:5522 (5.5 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

此时看到容器内的 IP 是 172.17.0.3, 接下来退出容器。

注意, 不要使用 exit 命令 (或者 Ctrl + C), 那样就是让 Docker 容器停止了。此时要退出容器, 可使用 Ctrl + P 命令, 然后使用 Ctrl + Q 命令, 即可退出容器的虚拟终端, 此

时容器还在运行。

另外注意一点，官方不推荐使用 `docker attach` 命令进入开启了交互模式的容器，即上面那样使用了 `-t` 参数的容器，`docker attach` 的主要功能是查看信息，容器内部操作有更加方便的 `docker exec` 命令，将在稍后介绍。

4.1.2 构建镜像的 `docker build` 命令

`docker build` 是构建镜像用的命令，将在第5章详细介绍，这里略讲部分，更有意思的内容在第5章。

从 `docker build` 的帮助信息中看到，`build` 这个子命令的功能非常强大。通过丰富的参数设置，可以控制镜像构建的各项细节。常用的参数命令主要如下。

- `-c`: 控制 CPU 使用。
- `-f`: 选择 Dockerfile 名称。
- `-m`: 设置构建内存上限。
- `-q`: 不显示构建过程的一些信息。
- `-t`: 为构建的镜像打上标签。

```
$ docker build --help
```

```
Usage: docker build [OPTIONS] PATH | URL | -
```

```
Build an image from a Dockerfile
```

```
Options:
```

```
  --build-arg value      Set build-time variables (default [])
  --cgroup-parent string Optional parent cgroup for the container
  --cpu-period int       Limit the CPU CFS (Completely Fair Scheduler)
period
  --cpu-quota int        Limit the CPU CFS (Completely Fair Scheduler)
quota
  -c, --cpu-shares int   CPU shares (relative weight)
  --cpuset-cpus string   CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems string   MEMs in which to allow execution (0-3, 0,1)
  --disable-content-trust Skip image verification (default true)
  -f, --file string      Name of the Dockerfile (Default is
'PATH/Dockerfile')
  --force-rm             Always remove intermediate containers
  --help                Print usage
  --isolation string     Container isolation technology
  --label value         Set metadata for an image (default [])
  -m, --memory string    Memory limit
  --memory-swap string   Swap limit equal to memory plus swap: '-1'
to enable unlimited swap
  --no-cache             Do not use cache when building the image
  --pull                Always attempt to pull a newer version of the
image
  -q, --quiet           Suppress the build output and print image ID
on success
  --rm                  Remove intermediate containers after a
successful build (default true)
  --shm-size string     Size of /dev/shm, default value is 64MB
  -t, --tag value       Name and optionally a tag in the 'name:tag'
format (default [])
```

`--ulimit value``Ulimit options (default [])`

有英文阅读能力的读者可以通过官方文档阅读有关 `build` 的内容，网址为 <https://docs.docker.com/engine/reference/commandline/build/>

4.1.3 提交容器的 `docker commit` 命令

`docker commit` 命令的主要功能是把当前容器提交打包为镜像，这一点可以联想到 Git 的操作，事实上 Docker 镜像就是使用 Git 式管理。

```
$ docker commit --help
Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
Create a new image from a container's changes
Options:
  -a, --author string      Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
  -c, --change value       Apply Dockerfile instruction to the created image (default [])
  --help                  Print usage
  -m, --message string     Commit message
  -p, --pause              Pause container during commit (default true)
```

把 `docker commit` 作为镜像构建的一种方式不失为一种办法，但是一般情况下，更加推荐使用 Dockerfile 构建镜像。

`docker commit` 的用法非常简单，主要参数如下。

- `-a`: 添加作者信息，方便维护。
- `-c`: 修改 Dockerfile 指令，目前支持的有以下指令。

`CMD` | `ENTRYPOINT` | `ENV` | `EXPOSE` | `LABEL` | `ONBUILD` | `USER` | `VOLUME` | `WORKDIR`

- `-m`: 类似 `git commit -m` 这样，提交修改信息。
- `-p`: 暂停正在 `commit` 的操作。

4.1.4 复制文件到宿主机的 `docker cp` 命令

`docker cp` 的作用就如名字，和 Linux 上著名的 `cp` 命令作用一样，用于在宿主机和容器之间移动复制文件。

```
$ docker cp --help
Usage: docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-
       Docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH
Copy files/folders between a container and the local filesystem
Options:
  -L, --follow-link  Always follow symbol link in SRC_PATH
  --help            Print usage
```

`docker cp` 命令非常简单，用一行命令就可以解释。

```
$ docker cp <containerId>:/file/path/within/container /host/path/target
```

举个例子，如要把容器的 `~/c_test.txt` 文件复制到宿主机，首先在宿主机上创建一个文件如下：

```
$ echo 这里是宿主机文件 > h_test.txt
$ ls
h_test.txt
```

然后创建一个测试容器。

```
$ docker run -idt --name ubuntu ubuntu:14.04 bash
2ec3c5a455e28e0b669f5a42b76f874e3be363053fce17620fd5522e471d3f6e
```

启动一个叫 **ubuntu** 的容器，并运行 **bash**，然后使用 **docker exec** 命令进入容器（不懂 **docker exec** 没关系，不是该例的重点）。

```
$ docker exec -it ubuntu bash
root@2ec3c5a455e2:/# cd home
root@2ec3c5a455e2:/home# pwd
/home
```

容器内进入 **home** 目录，并创建一个 **c_test.txt**。

```
root@2ec3c5a455e2:/home# echo "Here is container." > c_test.txt
root@2ec3c5a455e2:/home# ls
c_test.txt
```

使用 **docker cp** 命令从容器内部复制文件到宿主机。

```
$ docker cp ubuntu:/home/c_test.txt ~/c_test.txt
$ ls
c_test.txt h_test.txt
$ cat c_test.txt
Here is container.
```

从宿主机复制文件到容器方法就很多了，可以用数据卷，也可以用 Linux 发行版自带的 **cp** 命令，短小的内容也可以直接使用 **echo** 命令写入到指定文件中。但这部分内容不属于 **Docker** 命令的一部分，此处不做介绍，有兴趣的读者可以在网络上寻找资料，地址为 <http://stackoverflow.com/questions/22907231/copying-files-from-host-to-docker-container>。

4.1.5 创建容器的 **docker create** 命令

在 **Docker** 容器状态中有一种是 **Created**，表示容器已经创建，但是没有启动，它和 **Stop** 不同，**Stop** 通常都是手动或者外部操作容器停止的，而 **Created** 有可能是手动创建但是没有成功启动。

例如现在宿主机存在一个容器使用了 80 端口，然后再启动一个容器使用 80 端口时，就会出现容器启动失败，显示 **Created** 状态，表示容器创建成功，但因为某些原因无法运行，**Created** 状态的容器不占用内存和 CPU 资源。

```
$ docker create --help
```

```
Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

```
Create a new container
```

```
.....
```

```
-w, --workdir string
```

```
Working directory inside the container
```

从帮助命令中可以看到，**docker create** 和 **docker run** 命令非常相似。

创建后的容器可以使用 **docker start containerID** 的方式启动容器。通常使用 **docker create** 命令的场景是为接下来的容器启动做前置准备，应用场景比较少，因为有其他编排工具接管这项工作，这会在后面介绍。

4.1.6 查看容器变化的 docker diff 命令

Docker 提供了一个非常强大的命令 `diff`，可以列出容器内发生变化的文件和目录，变化包括添加（A-add）、删除（D-delete）、修改（C-change）。该命令便于 Debug，并支持快速的共享环境。

```
$ docker diff --help
Usage: docker diff CONTAINER
Inspect changes on a container's filesystem
Options:
    --help    Print usage
```

`docker diff` 的语法如下：

```
$ docker diff containerID
```

该命令同样让人想到 `git diff` 和 `git status` 命令，类似的，`docker diff` 命令主要用于显示当前运行容器和镜像的不同。

这里使用上面 `attach` 命令中运行的容器 `ubuntu` 作为例子：

```
$ docker diff ubuntu
C /home
A /home/c_test.txt
```

`docker diff` 的运行与容器的状态无关，只是显示文件差异。

4.1.7 查看事件的 docker events 命令

`docker events` 命令实时输出 Docker 服务器端的事件，包括容器的创建、启动、关闭等。

```
$ docker events --help
Usage: docker events [OPTIONS]

Get real time events from the server

Options:
  -f, --filter value  Filter output based on conditions provided (default [])
  --help              Print usage
  --since string       Show all events created since timestamp
  --until string       Stream events until this timestamp
```

下面启动两个终端，在第一个终端输入 `docker events`，然后在第二个终端启动容器 `ubuntu`，然后回到第一个终端可以看到启动信息。

```
$ docker start ubuntu
ubuntu
$ docker events
2016-09-05T15:54:24.368971374+08:00 network connect
2359c2167ba96123f6a36040bb50100d378f447cf683813d6e2dlb74dfb2aee7
(container=2ec3c5a455e28e0b669f5a42bh76f874e3be363053fcel7620fd5522e471
d3f6e, name=bridge, type=bridge)
2016-09-05T15:54:24.715464704+08:00 container start
2ec3c5a455e28e0b669f5a42b76f874e3be363053fcel7620fd5522e471d3f6e
(image=ubuntu:14.04, name=ubuntu)
```

`docker events` 涵盖了几乎全部 Docker 事件，通过 `-f` 指定参数，还可以过滤不必要的事

件, 得到更精简的事件信息。

使用说明:

```
$ docker events -f container=<name or id>
```

通过指定容器 ID 可以过滤其他信息, 和容器相关的事件有 attach, commit, copy, create, destroy, detach, die, exec_create, exec_detach, exec_start, export, kill, oom, pause, rename, resize, restart, start, stop, top, unpause, update。

```
$ docker events -f image=<tag or id>
```

通过指定镜像 ID 可以过滤其他信息, 和镜像相关的事件有 delete, import, load, pull, push, save, tag, untag。

```
$ docker events -f volume=<name or id>
```

通过指定 volume ID 可以过滤其他信息, 和 volume 相关的事件有 create, mount, unmount, destroy。

```
$ docker events -f network=<name or id>
```

通过指定网络 ID 可以过滤其他信息, 和网络相关的事件有 create, connect, disconnect, destroy。

```
$ docker events -f daemon=<name or id>
```

只有 reload 一个值, 用于监控记录 Docker 守护进程的状态。

其他还有:

```
$ docker events -f label=<key> or label=<key>=<value>
```

```
$ docker events -f event=<event action>
```

```
$ docker events -f type=<container or image or volume or network or daemon>
```

部分试验性功能没有加进来, 主要就是这些。

4.1.8 进入容器的 docker exec 命令

docker exec 主要是用于进入容器内部进行操作的一个重要命令, 比 attach 功能更强大, 通过 docker exec 可以像使用 SSH 登录服务器一样操作容器, 所以一般进入容器的命令都是使用 docker exec 而不是 docker attach。

```
$ docker exec -h
```

```
Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

```
Run a command in a running container
```

```
-d, --detach           Detached mode: run command in the background
--detach-keys          Override the key sequence for detaching a container
--help                Print usage
-i, --interactive      Keep STDIN open even if not attached
--privileged           Give extended privileges to the command
-t, --tty              Allocate a pseudo-TTY
-u, --user             Username or UID (format: <name|uid>[:<group|gid>])
```

docker exec 的常用参数有以下几个。

- -d 分离模式: 在后台运行的命令。

- -i 交互模式。
- -t 分配一个 TTY。
- -u 指定用户和用户组，格式：<name|uid>[:<group|gid>]。

例如，以上面的 ubuntu 容器为例，先启动容器，然后使用 exec 进入容器。

```
$ whoami
ubuntu
$ docker exec -it ubuntu bash
root@2ec3c5a455e2:/# whoami
root
root@2ec3c5a455e2:/#
```

可以看到使用 exec 命令进入容器内部就如同进入另一台机器一样，可以灵活操作，并且使用 exit 命令退出时，不会像 attach 那样导致容器停止，所以非常适合在容器内部操作。此外，每个 docker exec 命令都会分配一个不同的 tty 给用户，所以不会有冲突。

4.1.9 导出容器的 docker export 命令

docker export 命令用于导出本地存储的容器，和 docker save 类似(用于导出本地镜像)，docker export 导出的容器通常为 tar 包，方便传输到其他地方。

```
$ docker export --help
Usage: docker export [OPTIONS] CONTAINER
Export a container's filesystem as a tar archive
Options:
  --help            Print usage
  -o, --output string Write to a file, instead of STDOUT
```

使用方法有两种，第一种是使用 Docker 提供的 -o 参数指定目标文件位置和名称，第二种方法是使用 > 符号指向目标文件。例如：

```
$ docker export -o containerName.tar containerName
$ docker export containerName > containerName.tar
```

使用 docker export 命令导出的容器并不会压缩容器大小。

4.1.10 查看镜像历史的 docker history 命令

docker history 命令用于显示镜像的历史，可以查看镜像的历史变化。

```
$ docker history --help
Usage: docker history [OPTIONS] IMAGE
Show the history of an image
Options:
  --help            Print usage
  -H, --human       Print sizes and dates in human readable format (default true)
  --no-trunc        Don't truncate output
  -q, --quiet       Only show numeric IDs
```

下面以 ubuntu:14.04 为例，查看镜像历史。

```
$ docker history ubuntu:14.04
IMAGE          CREATED          CREATED BY                                      SIZE    COMMENT
4a725d3b3b1c   9 days ago      /bin/sh -c #(nop) CMD ["/bin/bash"]          0 B
<missing>      9 days ago      /bin/sh -c mkdir -p /run/systemd && echo 'doc
7 B
```



```

<missing>      9 days ago    /bin/sh -c sed -i 's/^#\s*\ (deb.*universe\)$ /
1.895 kB
<missing>      9 days ago    /bin/sh -c rm -rf /var/lib/apt/lists/*
0 B
<missing>      9 days ago    /bin/sh -c set -xe  && echo '#!/bin/sh' > /u
194.6 kB
<missing>      9 days ago    /bin/sh -c #(nop) ADD file:ada91758a31d8de3c7
187.7 MB

```

可以看到，镜像历史在一定程度反映了 Dockerfile 的内容。

4.1.11 查看本地镜像的 docker images 命令

docker images 命令用于查看本地存储的 Docker 镜像。

```
$ docker images --help
```

```
Usage: docker images [OPTIONS] [REPOSITORY[:TAG]]
```

```
List images
```

```
Options:
```

```

-a, --all           Show all images (default hides intermediate images)
--digests          Show digests
-f, --filter value  Filter output based on conditions provided (default [])
--format string     Pretty-print images using a Go template
--help             Print usage
--no-trunc         Don't truncate output
-q, --quiet        Only show numeric IDs

```

主要参数如下。

- -a: 显示所有镜像，包括中间镜像（悬挂镜像），默认不显示。
- -f: 过滤显示，可选的值有。

```
$ docker images -f dangling=[true|false]
```

```
$ docker images -f label=<key>[=<value>]
```

```
$ docker images -f before=(<image-name>[:tag]|<image-id>|<image@digest>)
```

```
$ docker images -f since=(<image-name>[:tag]|<image-id>|<image@digest>)
```

- -q: 只显示 ID。

使用 docker images ubuntu 可以查看和 Ubuntu 有关的本地镜像。下面以删除本地所有 none 镜像为例，使用 docker images 的 -f 参数。

```
$ docker images --filter "dangling=true"
```

```
REPOSITORY    TAG        IMAGE ID      CREATED      SIZE
<none>        <none>     5cabe3e4656d  42 hours ago  72.05 MB
```

```
<none>        <none>     97482528bdb9  42 hours ago  34.55 MB
```

```
<none>        <none>     462fca6e7913  42 hours ago  142.49 MB
```

```
$ docker rmi $(docker images -f "dangling=true" -q)
```

```
Deleted: sha256:5cabe3e4656d1f9541660adf10fe9874b49fd132389fe32611799551
1f2f8b6b
```

```
Deleted: sha256:f2b76b811484e855861888e0b9890d37c6ae9c6309ddd038fc2c53d3
614876c3
```

```
Deleted: sha256:45bf753ea7a25ae924a7424d6f6c83b058fc034acaf7a04880ddaf7d
ee9d3f21
```

```
Deleted: sha256:539688a4503833791d9fb5633925a62c5d3802fd82fd104860a9b32c
c4d03f52
```

```
Deleted: sha256:031143c1c662878cf5be0099ff759dd219f907a22113eb60241251d2
9344bb96
```

```
Deleted: sha256:9e63c5bce4585dd7038d830a1f1f4e44cb1a1515b00e620ac718e934
b484c938
```

docker images 命令的具体用法还可以很灵活，详见第 5 章内容。

4.1.12 导入容器的 docker import 命令

docker import 命令和 4.1.9 节的 docker export 命令相对，用于导入容器，导入后会变成镜像，用法和 docker export 命令相似。

```
$ docker import --help
```

```
Usage: docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

Import the contents from a tarball to create a filesystem image

Options:

```
-c, --change value      Apply Dockerfile instruction to the created image
                        (default [])
```

```
--help                  Print usage
```

```
-m, --message string    Set commit message for imported image
```

可以使用网络地址直接导入。

```
$ docker import https://example.com/container.tar
```

也通过管道导入。


```
$ cat exampleContainer.tgz | docker import --message "New image imported from
tarball" - exampleContainerlocal:newtag
```

还可以直接导入本地 tar 包。

```
$ docker import /path/to/exampleContainer.tgz
```

甚至可以从目录导入。

```
$ sudo tar -c . | docker import - exampleContainerdir
```

 **注意：**在这个例子中特别使用了 sudo，是因为在非 root 环境下直接导入一个目录有可能不能保留原有文件的权限属性，所以使用 docker import 命令导入需添加权限。

此外，在导入过程中使用--change 还可以改变 Dockerfile 中的指令，使用--message 可以添加 commit 信息。

4.1.13 查看 Docker 信息的 docker info 命令

docker info 大概是 Docker 的 45 个子命令中最简单易懂的命令了，输入 docker info 命令可以很详尽地看到 Docker 的各项信息（容器数量、状态，镜像数量，服务版本，存储驱动、根目录，数据卷，插件，网络，安全，硬件等信息）。

```
$ docker info
Containers: 4
Running: 4
Paused: 0
Stopped: 0
Images: 8
Server Version: 1.12.1
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
```

```

Dirs: 44
Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge null host overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: apparmor seccomp
Kernel Version: 4.4.0-36-generic
Operating System: Ubuntu 16.04.1 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 16 GB
Name: ubuntu
ID: XKPV:JQOO:9MES:A9DI:42LQ:PKRY:T24V:2QXI:YFFT:VNHI:7FWZ:ZQNB
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Username: ubuntu
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
127.0.0.0/8

```

通过 `docker info` 命令可以查看 Docker 运行状态，除了直接运行 `docker info` 命令之外，还可以加上 `-D` 参数显示 Docker system 的 Debug 信息。

4.1.14 查看各项详细信息的 `docker inspect` 命令

`inspect` 有检查、检阅的意思，所以 `docker inspect` 命令的用途是检查容器或者镜像详细信息的一个命令，这个命令用法虽简单，输出的内容却相当丰富，下面对这个命令输出做部分解读，更详细的部分在后续容器和镜像相关的章节中还会介绍。

```
$ docker inspect -h
```

```
Usage: docker inspect [OPTIONS] CONTAINER|IMAGE|TASK [CONTAINER|IMAGE|TASK...]
```

```
Return low-level information on a container, image or task
```

```

-f, --format          Format the output using the given go template
--help               Print usage
-s, --size            Display total file sizes if the type is container
--type               Return JSON for specified type, (e.g image, container
or task)

```

从上面提示可以看出 `docker inspect` 是一个查看镜像、容器运行时详细信息的命令。了解一个 Image 或者 Container 的完整构建信息就可以通过该命令实现。

下面继续用上面创建的 `ubuntu` 容器为例子，使用 `docker inspect` 命令查看 `ubuntu` 容器的相关信息，因为输出信息较多，而 `docker inspect` 命令在后面还会用到，因此这里不贴完整的输出信息。

查看 ubuntu 容器的 IP 地址。

```
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}
{{end}}' ubuntu
172.17.0.3
```

查看 ubuntu 容器的 Mac 地址。

```
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.MacAddress}}
{{end}}' ubuntu
02:42:ac:93:00:04
```

查看 ubuntu 容器的日志。

```
$ docker inspect --format='{{.LogPath}}' ubuntu
/var/lib/docker/containers/2ec3c5a455e28e0b669f5a42b76f874e3be363053fce
17620fd5522e471d3f6e/2ec3c5a455e28e0b669f5a42b76f874e3be363053fce17620f
d5522e471d3f6e-json.log
```

还有其他用法会在文后逐渐接触。

4.1.15 杀死容器的 docker kill 命令

使用 `docker kill` 命令可以快速杀掉容器的进程，常用在无法停止容器的时候，使用 `docker kill` 命令可以让一个无响应的容器停止。

`stop` 和 `kill` 的区别在于 `docker stop` 命令给容器中的进程发送 `SIGTERM` 信号，默认行为是会导致容器退出，当然，容器内程序可以捕获该信号并自行处理，例如可以选择忽略。而 `docker kill` 命令则是给容器的进程发送 `SIGKILL` 信号，该信号将会使容器必然退出。

```
$ docker kill --help
Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]
Kill one or more running containers
Options:
  --help            Print usage
  -s, --signal string Signal to send to the container (default "KILL")
```

`docker kill` 命令可以使用 `--signal` 指定发送的信号，信号是 UNIX、类 UNIX 以及其他 POSIX 兼容的操作系统中进程间通信的一种有限制的方式。

4.1.16 导入镜像的 docker load 命令

`docker load` 命令的作用是导入使用 `docker save` 导出的镜像，和 `export`、`import` 负责容器的导出导入类似，`save`、`load` 负责镜像的导出导入。

```
$ docker load --help
Usage: docker load [OPTIONS]
Load an image from a tar archive or STDIN
Options:
  --help            Print usage
  -i, --input string Read from tar archive file, instead of STDIN
  -q, --quiet        Suppress the load output
```

`-i` 指定导出文件，例如：

```
$ docker load -i ubuntu.tar
```

`-q` 可以不显示导入的一些信息，在一些脚本中可以有效减少输出干扰。

还可以通过“<”符号导入，下面以一个之前导出的镜像 busybox 为例，进行说明。

```
$ docker load < busybox.tar.gz
# [...]
Loaded image: busybox:latest
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
busybox	latest	769b9341d937	7 weeks ago	2.489 MB

4.1.17 登录仓库的 docker login 命令

docker login 命令是一个登录到 Registry 的命令，Registry 是 Docker 公司为了方便镜像流通而设计的一种镜像仓库，像手机上的应用商店一样，用户可以在上面发布镜像和拉取镜像，官方的 Docker Hub 还提供更高级的企业服务。

```
$ docker login --help
Usage: docker login [OPTIONS] [SERVER]
Log in to a Docker registry.
Options:
  --help            Print usage
  -p, --password string Password
  -u, --username string Username
```

使用 docker login 命令直接登录到 Docker Hub。

```
$ docker login
Username: username
Password:
Login Succeeded
```

使用 docker login localhost:8080 的方式可以登录到第三方仓库。

登录信息均会保存在 \$HOME/.docker/config.json 目录下，如果用户使用第三方证书存储，可以参考下面链接。

- D-Bus Secret Service 的网址为 <https://github.com/docker/docker-credential-helpers/releases>。
- Apple OS X keychain 的网址为 <https://github.com/docker/docker-credential-helpers/releases>。
- Microsoft Windows Credential Manager 的网址为 <https://github.com/docker/docker-credential-helpers/releases>。

然后在 config.json 中修改：

```
{
  "credsStore": "osxkeychain"
}
```

4.1.18 登出仓库的 docker logout 命令

docker logout 命令显而易见是登出命令，使用该命令可以登出仓库。如果使用第三方证书存储，只需要删除证书文件以及 config.json 即可。

4.1.19 查看容器日志的 docker logs 命令

`docker logs` 命令的功能是用于显示容器的日志，这一点和 `attach` 有点类似，不同的地方在于 `attach` 可以相对输出更自由，用户可以定制输出内容，而 `logs` 则是根据容器命令输出信息，是无交互的。

```
$ docker logs --help
Usage: docker logs [OPTIONS] CONTAINER
Fetch the logs of a container
Options:
  --details          Show extra details provided to logs
  -f, --follow       Follow log output
  --help            Print usage
  --since string     Show logs since timestamp
  --tail string      Number of lines to show from the end of the logs (default
                    "all")
  -t, --timestamps  Show timestamps
```

主要参数如下。

- `--details`: 显示更详细的日志。
- `-f`: 持续输出日志。
- `--since <string>`: 显示某字符串开始的日志。
- `--tail <string>`: 显示某字符串之前的日志。
- `-t`: 显示时间戳。

```
$ docker logs ubuntu
root@2ec3c5a455e2:/# exit
root@2ec3c5a455e2:/# exit
$ docker logs -t ubuntu
2016-09-05T07:49:41.615821786Z root@2ec3c5a455e2:/# exit
2016-09-05T08:04:40.658702755Z root@2ec3c5a455e2:/# exit
$ docker logs -f ubuntu
# 使用这个参数会持续输出而不会截断信息
```

4.1.20 管理网络的 docker network 命令

`docker network` 命令有点特殊，如果说 `network` 是 Docker 的子命令，那么 `network` 还有几个自己的子命令。所以 `network` 子命令功能非常强大，而且更新非常频繁，改动很大，详细用法会在网络相关章节讲解。

```
$ docker network --help
Usage: docker network COMMAND
Manage Docker networks
Options:
  --help  Print usage
Commands:
  connect  Connect a container to a network
  create   Create a network
  disconnect Disconnect a container from a network
  inspect  Display detailed information on one or more networks
  ls       List networks
  rm       Remove one or more networks
Run 'docker network COMMAND --help' for more information on a command.
```


关于网络管理部分在前面几章都不会涉及，所以这里只简单介绍，在后面实战部分会详细展开。如表 4.1 所示为 docker network 命令说明。

表 4.1 docker network 命令说明

命 令	说 明
network connect	连接一个容器到指定网络
network create	创建一个网络
network disconnect	指定网络断开一个容器
network inspect	显示指定网络详细信息
network ls	显示全部Docker 网络
network rm	删除指定网络

在 network 命令出现之前，用户只能手动创建管理网络，network 命令发布之后这一切变得简单起来，例如使用 ls 命令可以查看本机的容器网络。

```
$ docker network ls
NETWORK ID      NAME                DRIVER             SCOPE
b61c6f0f8ce7    bridge              bridge              local
db8e9c87359c    docker_gwbridge     bridge              local
fbd9a977aa03    host                host                local
1082eea7d0eb    nginx_default       bridge              local
e81af24d643d    none                null                local
```

4.1.21 管理节点的 docker node 命令

和前面一样，docker node 子命令同样有很多子命令，它是一个集群管理的命令，使用该命令可以轻松管理集群。该命令是 Docker 1.12 加进来的新子命令，为了更容易理解 node 和 swarm 的概念，将这部分放到了本书的 Docker Swarm 章节，在讲到 Docker 的生态系统时再详细解释。

```
$ docker node --help
Usage: docker node COMMAND
Manage Docker Swarm nodes
Options:
    --help    Print usage
Commands:
    demote    Demote one or more nodes from manager in the swarm
    inspect   Display detailed information on one or more nodes
    ls        List nodes in the swarm
    promote   Promote one or more nodes to manager in the swarm
    rm        Remove one or more nodes from the swarm
    ps        List tasks running on a node
    update    Update a node
Run 'docker node COMMAND --help' for more information on a command.
```

这里的集群、节点的概念涉及更复杂的内容，如果有兴趣可以阅读本书 docker swarm 的相关内容。

如表 4.2 所示为 docker node 的命令说明，该命令的有些功能只有在 manager 节点才能执行。

表 4.2 docker node的命令说明

命 令	说 明
node promote	提升节点为manager节点

(续)

命 令	说 明
node demote	将集群中的指定manager节点降权
node inspect	显示节点的详细信息
node update	更新节点属性
node ps	显示正在运行的节点
node ls	显示集群的全部节点
node rm	从集群中删除指定节点

4.1.22 暂停容器的 docker pause 命令

docker pause 命令会暂停容器内的所有进程，此时，通过 docker stats 可以观察到此时的资源使用情况是固定不变的，通过 docker logs -f 也观察不到日志的进一步输出。

该命令会使用 cgroup 的 freezer 顺序暂停容器里的所有进程。

```
$ docker pause --help
Usage: docker pause CONTAINER [CONTAINER...]
Pause all processes within one or more containers
Options:
    --help    Print usage
```

命令格式如下：

```
$ docker pause <container>
```

4.1.23 查看容器端口的 docker port 命令

docker port 命令用来输出容器的端口信息，与 Docker ps 的显示不同，该命令只会显示“暴露”的端口，对于未指定的未暴露端口不会显示。

```
$ docker port --help
Usage: docker port CONTAINER [PRIVATE_PORT[/PROTO]]
List port mappings or a specific mapping for the container
Options:
    --help    Print usage
```

例如，使用 ps 显示容器 test 的端口。

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  PORTS                  NAMES
b650456536c7   busybox:latest  top                    0.0.0.0:1234->9876/tcp, 0.0.0.0:4321->7890/tcp  test
7544d72cd84e   nginx:alpine    nginx                  80/tcp, 443/tcp, 0.0.0.0:2015->2015/tcp  nginx
```

使用 port 显示的端口。

```
$ docker port test
7890/tcp -> 0.0.0.0:4321
9876/tcp -> 0.0.0.0:1234
$ docker port test 7890/tcp
0.0.0.0:4321
$ docker port test 7890/udp
2014/06/24 11:53:36 Error: No public port '7890/udp' published for test
```

```
$ docker port test 7890
0.0.0.0:4321
```

可以看到 `port` 的信息比 `ps` 清晰，除此之外，像 Nginx 容器并未暴露 80 和 443 端口到宿主机，所以使用 `port` 不会显示以下信息：

```
$ docker port nginx
2015/tcp -> 0.0.0.0:2015
```

4.1.24 查看本地容器信息的 docker ps 命令

`docker ps` 命令显示当前正在运行的容器，通过相关参数配合 Linux 命令可以实现灵活的操作。

```
$ docker ps --help
Usage: docker ps [OPTIONS]
List containers
Options:
  -a, --all                Show all containers (default shows just running)
  -f, --filter value       Filter output based on conditions provided (default [])
  --format string          Pretty-print containers using a Go template
  --help                   Print usage
  -n, --last int           Show n last created containers (includes all states) (default -1)
  -l, --latest             Show the latest created container (includes all states)
  --no-trunc              Don't truncate output
  -q, --quiet              Only display numeric IDs
  -s, --size               Display total file sizes
```

主要的参数如下。

- `-a`: 显示全部容器，包括各种状态的容器，只要存在就显示。
- `-f`: 添加过滤条件。
- `-n`: 显示最近创建的几个容器（包括所有状态的容器，`-l` 显示最近创建的一个，遇到持续输出的信息还可以使用 `--no-trunc` 进行追加）。
- `-q`: 只显示 ID。
- `-s`: 显示容器大小。

使用 `ps` 命令可以实现很多便捷的操作，例如下面的一句命令删除所有已经停止的容器。

```
$ docker rm $(docker ps -a -q)
```

甚至还可以删除所有容器（包括运行中的容器）：

```
$ docker kill $(docker ps -q) ; Docker rm $(docker ps -a -q)
```

以及杀死所有正在运行的容器：

```
$ docker kill $(docker ps -a -q)
```

4.1.25 拉取镜像的 docker pull 命令

`docker pull` 命令为拉取镜像的命令，通过该命令不仅可以拉取 Docker Hub 的镜像，还

可以通过指定仓库地址拉取私有仓库镜像。

```
$ docker pull --help
Usage: docker pull [OPTIONS] NAME[:TAG|@DIGEST]
Pull an image or a repository from a registry
Options:
  -a, --all-tags          Download all tagged images in the repository
  --disable-content-trust Skip image verification (default true)
  --help                  Print usage
```

使用 `docker pull -a` 会把所有标签都拉取到本地，使用 `--disable-content-trust=false` 会在拉取时校验镜像，保证传输安全，默认是关闭的。

4.1.26 推送镜像的 docker push 命令

`docker push` 命令的作用是把本地的镜像推送到镜像仓库，和 `docker pull` 一样，使用 `--disable-content-trust=false` 命令会在拉取时校验镜像，保证传输安全，默认是关闭的。

```
$ docker push --help
Usage: docker push [OPTIONS] NAME[:TAG]
Push an image or a repository to a registry
Options:
  --disable-content-trust Skip image verification (default true)
  --help                  Print usage
```

使用 `docker push` 命令时，如果不指定 `tag` 会默认把该镜像的全部镜像都推送到仓库。例如，本地存在 `ubuntu:14.04` 和 `ubuntu:16.06` 两个镜像，如果使用 `docker push ubuntu` 命令推送会把这两个镜像都推送到仓库。

4.1.27 重命名容器的 docker rename 命令

`docker rename` 命令的作用是重命名容器，该命令可以在不改变容器状态的情况下重命名容器。格式是：`docker rename <旧容器名> <新容器名>`，该命令一次只能更改一个容器名称。

```
$ docker rename --help
Usage: docker rename OLD_NAME NEW_NAME
Rename a container
Options:
  --help  Print usage
```

这一点在为运行中的容器改名时非常有用。该命令只能用在容器上，镜像重命名可以使用后面会提到的 `docker tag` 命令。

4.1.28 重启容器的 docker restart 命令

`docker restart` 命令的作用是重启容器（不是重启 Docker，重启 Docker 可以使用 `systemctl restart docker` 命令或者 `service docker restart` 命令重启 Docker）。使用 `docker restart <Container1> <Container2>` 命令可以重启多个容器。

```
$ docker restart --help
Usage: docker restart [OPTIONS] CONTAINER [CONTAINER...]
```

Restart a container

Options:

```
--help      Print usage
-t, --time int  Seconds to wait for stop before killing the container
(default 10)
```

通过添加 `-t` 参数，可以在重启时设置等待容器停止的时间，如果容器在指定秒数之内没有停止，Docker 就会执行 `docker kill` 操作杀死容器，以便完成重启操作。

实际上 `docker restart` 命令的整个过程就是 `docker stop` 命令加上 `docker start` 命令。

4.1.29 删除容器的 `docker rm` 命令

`docker rm` 是一个删除容器的命令，直接使用 `docker rm <Container Name / Container ID>` 命令可以删除已经停止的容器。该命令可以删除一个或者多个容器。

```
$ docker rm --help
Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]
Remove one or more containers
Options:
-f, --force      Force the removal of a running container (uses SIGKILL)
--help          Print usage
-l, --link       Remove the specified link
-v, --volumes   Remove the volumes associated with the container
```

在 `docker rm` 命令中，有 3 个参数，其中 `-f` 是比较常用的，可以直接删除一个正在运行的容器，如果是 `docker rm` 命令只能删除非运行状态的容器，添加了 `-f` 的参数之后，Docker 会向指定容器发送 `SIGKILL` 使其停止然后删除该容器；`-l` 参数的作用是删除容器与其他容器的关联，但会保留容器，这里涉及容器间的通信知识，在后面会讲到；`-v` 参数会删除容器的数据卷，这也是后面要讲到的内容，默认是不会删除数据卷的，数据卷是挂载到容器的一个目录，它与容器的生命周期独立，不会因为容器的销毁而消失，详细内容在第 8 章和第 9 章的内容中会讲到。

4.1.30 删除镜像的 `docker rmi` 命令

`docker rmi` 是一个删除镜像的命令，删除镜像时最好指定镜像的 `tag`，如果不指定会默认删除镜像的 `latest` 标签。该命令同样可以在后面接上多个镜像名称，删除多个镜像。

使用 `docker rmi` 命令删除镜像时，要确保没有容器在使用该镜像，也就是没有容器是使用该镜像启动的，才可以删除，否则会报错。

```
$ docker rmi --help
Usage: docker rmi [OPTIONS] IMAGE [IMAGE...]
Remove one or more images
Options:
-f, --force      Force removal of the image
--help          Print usage
--no-prune      Do not delete untagged parents
```

删除镜像时不一定要使用镜像名称，同 `docker rm` 命令一样，后面可以是镜像的 ID，`docker rmi` 命令还有一个参数 `-f`，该参数可以强制删除镜像，即便有容器正在使用该镜像，但是这样只会删除镜像标签，不影响正在运行的容器。实际上只要容器还在运行，镜像就不会被真正删除，用户使用 `docker commit` 操作提交容器为镜像，可以恢复镜像。

4.1.31 运行容器的 docker run 命令

docker run 命令的作用是启动容器，在整个 Docker 命令中可以说是功能最强大的一个子命令，这里只对 docker run 命令做简单介绍，在后面章节中会详细说明该命令的作用和原理。

当运行 docker run 命令时，Docker 会启动一个容器进程，并为这个进程分配其独占的文件系统、网络资源和以此进程为根进程的进程组。

在容器启动时，用户可以通过 docker run 命令重新定义镜像的配置，包括要运行程序、暴露的网络端口等，docker run 命令可以覆盖 docker build 命令在构建镜像时的一些默认配置，这也是为什么 run 命令相比于其他命令有如此多参数的原因。

为了方便阅读，下面会把 help 信息截断来介绍，加粗内容表示属于 help 的信息。

```
$ docker run --help
```

```
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

```
Run a command in a new container
```

```
Options:
```

```
--add-host value          Add a custom host-to-IP mapping (host:ip)
(default [])
```

--add-host 可以理解为动态地添加一行到映射/etc/hosts 文件中，格式是：

```
myWeb:172.16.233.233
```

这样在容器内部就会在/etc/hosts 中添加一行：

```
172.16.233.233 myWeb
```

再来看下面：

```
-a, --attach value        Attach to STDIN, STDOUT or STDERR (default [])
```

形如 docker attach 命令，可以选择的参数值是 STDIN, STDOUT, STDERR。在 docker run 命令中如果不使用 -d 参数（后台运行）运行容器，容器就会在前台模式下运行，将当前的命令行窗口附着到容器的标准输入、标准输出和标准错误中。也就是说容器中所有的输出都可以在当前窗口中看到，甚至可以虚拟出一个 TTY 窗口来执行信号中断。而这里的 -a 参数就是可以选择输出哪些信息，方便用户定位信息。注意，这里的 -a 不能与 -d 一起使用。

```
--blkio-weight value      Block IO (relative weight), between 10 and
1000
```

```
--blkio-weight-device value Block IO weight (relative device weight)
(default [])
```

上面两个是容器磁盘 I/O 限制的参数，众所周知，容器将会为用户提供一个隔离的运行环境，容器内部的进程或者进程组使用资源时需要受到限制，这样的资源包括内存资源（物理内存以及 swap），CPU 资源（CPU 时间片以及 CPU 核等），磁盘空间资源等。Docker 添加了一 blkio-weight 参数，实现对容器磁盘 I/O 限制的支持，用户也不需要再担心容器间磁盘 I/O 资源的竞争。

--cap-add value	Add Linux capabilities (default [])
--cap-drop value	Drop Linux capabilities (default [])

上面两个参数也是一对，`--cap-add` 可以添加容器对内核操作的权限，而 `--cap-drop` 可以降权，默认使用这两个参数的情况下，容器拥有一系列的内核修改权限，这两个参数都支持 `all` 值，如果想让某个容器拥有除了 `MKNOD` 之外的所有内核权限，那么可以执行下面的命令：

```
$ sudo docker run --cap-add=ALL --cap-drop=MKNOD ...
```

比这两个参数更高级权限的获取可以使用 `--privileged` 参数，但是像修改网络接口数据这种具体的事情，建议使用 `--cap-add=NET_ADMIN`，而不是使用 `--privileged` 获取全部权限。

--cgroup-parent string	Optional parent cgroup for the container
-------------------------------	---

上面命令为容器指定父级 cgroup，一般很少用到。

--cidfile string	Write the container ID to the file
-------------------------	---

如果在使用 Docker 时有自动化的需求，可以将 `containerID` 输出到指定的文件中（`PIDfile`），类似于某些应用程序将自身 ID 输出到文件中，方便后续脚本操作。

--cpu-percent int	CPU percent (Windows only)
--cpu-period int	Limit CPU CFS (Completely Fair Scheduler) period
--cpu-quota int	Limit CPU CFS (Completely Fair Scheduler) quota
-c, --cpu-shares int	CPU shares (relative weight)
--cpuset-cpus string	CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems string	MEMs in which to allow execution (0-3, 0,1)

- `--cpu-percent`: 设置 CPU 使用率的百分比，只有 Windows 下才能起效。
- `--cpu-period`: 用来指定容器对 CPU 的使用要在多长时间内做一次重新分配。
- `--cpu-quota`: 用来指定在这个周期内，最多可以有多少时间用来运行这个容器。
- `--cpu-shares`: 用来设置容器对 CPU 使用的权重，默认情况下所有容器的 `share`（理解为权重）是相同的，也就是所有容器有相同的权重，在所有容器一起竞争资源时，最终得到的资源是相同的。这个 `share` 是一个相对的值，比如 A 和 B 两个容器，A 配置的是 1024，B 配置的是 512，那么 A 最大可以使用的 CPU 资源是 B 的两倍。还有一点要注意的是这种配置是有弹性的，如果 A 容器一直闲着，那 B 容器是可以使用空闲资源的。
- `--cpuset-cpus`: 可以绑定指定容器使用指定 CPU。
- `--cpuset-mems`: 只应用于 NUMA 架构的 CPU 生效。

-d, --detach	Run container in background and print container ID
---------------------	---

`-d` 参数在 `docker run` 中比较常用，添加 `-d` 参数可以让容器在后台运行，返回容器 ID，使用 `docker ps` 命令可以查看正在运行的容器。

--detach-keys string	Override the key sequence for detaching a container
--device value	Add a host device to the container (default [])
--device-read-bps value	Limit read rate (bytes per second) from a device (default [])
--device-read-iops value	Limit read rate (IO per second) from a device (default [])
--device-write-bps value	Limit write rate (bytes per second) to a device (default [])

```

--device-write-iops value    Limit write rate (IO per second) to a
device (default [])
--disable-content-trust      Skip image verification (default true)
--dns value                  Set custom DNS servers (default [])
--dns-opt value              Set DNS options (default [])
--dns-search value           Set custom DNS search domains (default [])
--entrypoint string          Overwrite the default ENTRYPOINT of the
image

```

设置入口点，作用与 Dockerfile 中的 ENTRYPOINT 指令一致，详细见第 6 章的内容。

```

-e, --env value              Set environment variables (default [])

```

-e 参数的作用是在启动容器时，修改镜像中预设的环境变量。在 Docker 镜像构建过程中，允许使用 ENV 指令设置环境变量，该变量不会随镜像构建结束而消失，而是保存到镜像中，在启动容器时通过 -e 参数可以重新设置覆盖原有的环境变量。

```

--env-file value             Read in a file of environment variables
(default [])
--expose value               Expose a port or a range of ports (default
[])
--group-add value            Add additional groups to join (default [])
--health-cmd string          Command to run to check health
--health-interval duration   Time between running the check
--health-retries int         Consecutive failures needed to report
unhealthy
--health-timeout duration    Maximum time to allow one check to run
--help                       Print usage
-h, --hostname string        Container host name

```

-h 参数指定容器 host name，使用该参数相当于在 /etc/hosts 文件中添加如下一行：

```
127.0.0.1 <set_host_name>
```

再来看下面：

```

-i, --interactive            Keep STDIN open even if not attached

```

-i 参数会保持 STDIN 开放，即使没有使用 attach 命令，配合 -t 参数可以为用户提供一个可交互的终端。

```

--io-maxbandwidth string     Maximum IO bandwidth limit for the system
drive (Windows only)
--io-maxiops uint            Maximum IOps limit for the system drive
(Windows only)
--ip string                   Container IPv4 address (e.g. 172.30.100.104)
--ip6 string                  Container IPv6 address (e.g. 2001:db8::33)
--ipc string                  IPC namespace to use
--isolation string            Container isolation technology
--kernel-memory string        Kernel memory limit
-l, --label value             Set meta data on a container (default [])

```

-l 参数可以在容器启动时设置 label，使用 label 设置元数据有助于辨识容器。

```

--label-file value           Read in a line delimited file of labels
(default [])
--link value                  Add link to another container (default [])

```

--link 参数用于容器之间的连接通信，在第 10 章的网络管理中会介绍到该参数。

```

--link-local-ip value         Container IPv4/IPv6 link-local addresses
(default [])
--log-driver string           Logging driver for the container
--log-opt value               Log driver options (default [])

```

```

--mac-address string      Container MAC address (e.g. 92:d0:c6:0a:29:33)
-m, --memory string      Memory limit
--memory-reservation string Memory soft limit
--memory-swap string      Swap limit equal to memory plus swap: '-1'
to enable unlimited swap
--memory-swappiness int   Tune container memory swappiness (0 to 100)
(default -1)

```

-m 参数设置容器可用内存的最大值。

```

--name string              Assign a name to the container

```

--name 参数用于设置容器名称，如没有添加此参数，Docker 会自动为容器命名，在启动时设置容器名称有助于管理容器。该参数的值在容器启动后可以使用 `docker rename` 命令重命名。

```

--network string          Connect a container to a network (default
"default")
--network-alias value     Add network-scoped alias for the
container (default [])
--no-healthcheck          Disable any container-specified HEALTHCHECK
--oom-kill-disable        Disable OOM Killer
--oom-score-adj int       Tune host's OOM preferences (-1000 to
1000)
--pid string              PID namespace to use
--pids-limit int          Tune container pids limit (set -1 for
unlimited)
--privileged              Give extended privileges to this container

```

--privileged 参数可以使容器使用宿主机的管理员权限，这意味着使用该参数启动的容器内部的 root 账号，拥有宿主机 root 级别的权限，一般情况下不用该参数。可以通过上面讲到的权限管理参数调整容器权限。

```

-p, --publish value       Publish a container's port(s) to the host
(default [])
-P, --publish-all        Publish all exposed ports to random ports

```

-p 参数指定容器向宿主机的端口映射，例如 -p 8080:80 可以把容器内的 80 端口映射到宿主机的 8080 端口，相关内容在第 10 章介绍。-P 参数会暴露容器的全部端口，适合一些不确定端口以及端口数量庞大的容器。

```

--read-only               Mount the container's root filesystem as
read only

```

--read-only 参数可以设置容器内的文件系统为只读，从而达到保护宿主机文件系统的安全。

```

--restart string          Restart policy to apply when a container
exits (default "no")

```

--restart string 参数可以设置容器在意外停止时自动重启。默认是 --restart=no，--restart=always 表示自动重启，手动使用 `docker stop` 停止容器除外。

```

--rm                      Automatically remove the container when it
exits

```

--rm 参数可以在容器退出时自动删除该容器，通常用在演示、测试、编译等场景，--rm 不能与 -d 参数共存。

```

--runtime string          Runtime to use for this container
--security-opt value      Security Options (default [])

```

<code>--shm-size</code> string	Size of /dev/shm, default value is 64MB
<code>--sig-proxy</code> (default true)	Proxy received signals to the process
<code>--stop-signal</code> string default (default "SIGTERM")	Signal to stop a container, SIGTERM by default
<code>--storage-opt</code> value (default [])	Storage driver options for the container
<code>--sysctl</code> value	Sysctl options (default map[])
<code>--tmpfs</code> value	Mount a tmpfs directory (default [])
<code>-t, --tty</code>	Allocate a pseudo-TTY

-t 参数表示向容器申请一个 tty，允许用户输入操作，通常配合-i 参数，以提供一个交互终端界面。

<code>--ulimit</code> value	Ulimit options (default [])
<code>-u, --user</code> string	Username or UID (format: <name uid>[:<group gid>])
<code>--userns</code> string	User namespace to use
<code>--uts</code> string	UTS namespace to use

-u 参数指定容器运行时的默认用户，在 Docker 镜像构建过程中，会使用 root 用户作为默认用户，通过 Dockerfile 的 USER 指令可以切换构建时的用户，因为 docker run 命令允许对 Dockerfile 的指令覆盖修改，所以-u 参数可以重新指定用户。

<code>-v, --volume</code> value	Bind mount a volume (default [])
<code>--volume-driver</code> string	Optional volume driver for the container
<code>--volumes-from</code> value (default [])	Mount volumes from the specified container(s)

-v 参数用来挂载数据卷，格式可以是 -v /host_dir:/container_dir，也可以是 -v /container_dir 格式。指定的数据卷与容器的生命周期完全独立，不会因为容器删除而丢失。在第 9 章中会单独讲解数据卷的使用与原理。

<code>-w, --workdir</code> string	Working directory inside the container
-----------------------------------	--

-w 参数用于指定容器的工作目录，也就是命令执行时的目录，一般在构建镜像时会通过 WORKDIR 指定，但是在 docker run 命令中也可以根据需要覆盖原有的工作目录。

4.1.32 导出镜像的 docker save 命令

docker save 命令是用来导出镜像的，后面可以接多个镜像名称，默认输出的是 STDOUT，意味着需要指定一个目标文件，可以通过-o 参数来指定。

```
$ docker save --help
Usage: docker save [OPTIONS] IMAGE [IMAGE...]
Save one or more images to a tar archive (streamed to STDOUT by default)
Options:
  --help            Print usage
  -o, --output string Write to a file, instead of STDOUT
```

最常用的就是导出一个镜像。

```
$ docker save -o ubuntu.tar ubuntu:14.04
$ ls
$ ubuntu.tar
```

还可以一起导出多个容器。

```
$ docker save -o nginx_php.tar nginx:1.9 php:7.0-fpm
$ ls
$ nginx_php.tar
```

上面导出的文件 `nginx_php.tar` 里面包含了两个镜像，在使用 `docker load` 命令导入的时候，Docker 会自动识别并导入两个镜像。

除了使用 `-o` 参数，还可以使用 `>` 符号导出镜像：

```
$ docker save ubuntu:14.04 > ubuntu.tar
```

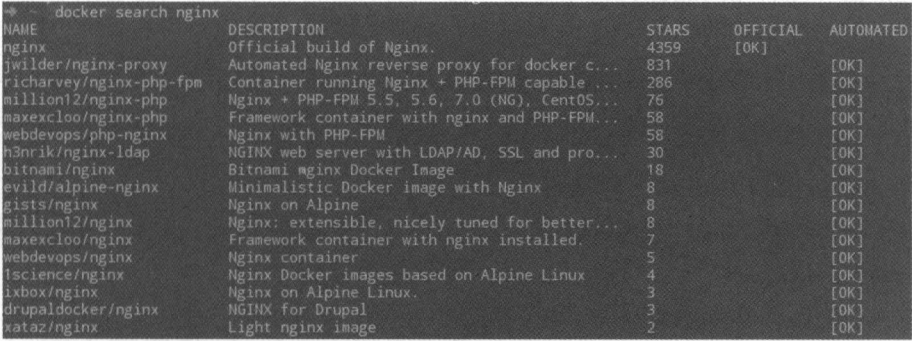
4.1.33 搜索镜像的 docker search 命令

`docker search` 命令很简单，可以使用关键字搜索分享的 Image。

```
$ docker search --help
Usage: docker search [OPTIONS] TERM
Search the Docker Hub for images
Options:
  -f, --filter value  Filter output based on conditions provided (default [])
  --help              Print usage
  --limit int         Max number of search results (default 25)
  --no-trunc          Don't truncate output
```

例如，使用 `docker search ubuntu` 命令搜索 Nginx 镜像，如图 4.1 所示。

```
$ docker search nginx
```



NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
nginx	Official build of Nginx.	4359	[OK]	
jwilder/nginx-proxy	Automated Nginx reverse proxy for docker c...	831		[OK]
richarvey/nginx-php-fpm	Container running Nginx + PHP-FPM capable ...	286		[OK]
million12/nginx-php	Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS...	76		[OK]
maxexcloo/nginx-php	Framework container with nginx and PHP-FPM...	58		[OK]
webdevops/php-nginx	Nginx with PHP-FPM	58		[OK]
h3nrik/nginx-ldap	NGINX web server with LDAP/AD, SSL and pro...	30		[OK]
bitnami/nginx	Bitnami nginx Docker Image	18		[OK]
evild/alpine-nginx	Minimalistic Docker image with Nginx	8		[OK]
gists/nginx	Nginx on Alpine	8		[OK]
million12/nginx	Nginx: extensible, nicely tuned for better...	8		[OK]
maxexcloo/nginx	Framework container with nginx installed.	7		[OK]
webdevops/nginx	Nginx container	5		[OK]
lscience/nginx	Nginx Docker images based on Alpine Linux	4		[OK]
ixbox/nginx	Nginx on Alpine Linux.	3		[OK]
drupaldocker/nginx	NGINX for Drupal	3		[OK]
xataz/nginx	Light nginx image	2		[OK]

图 4.1 搜索 Nginx 镜像

通过使用 `-f` 参数还可以定制返回信息，可选参数共有以下 3 个。

- `is-automated=(true|false)`;
- `is-official=(true|false)` ;
- `stars=<number>`。

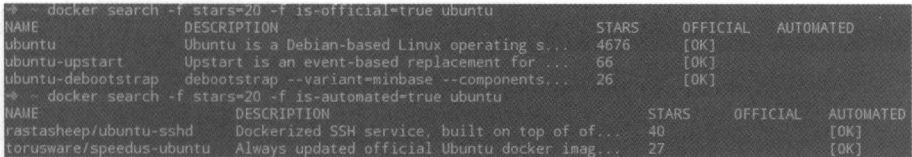
例如，搜索 Ubuntu 的官方镜像：

搜索 stars 超过 20 并且是官方构建的 Ubuntu 镜像：

```
$ docker search -f stars=20 -f is-official=true ubuntu
```

搜索 stars 超过 20 并且是自动构建的 Ubuntu 镜像，如图 4.2 所示。

```
$ docker search -f stars=20 -f is-automated=true ubuntu
```



NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
ubuntu	Ubuntu is a Debian-based Linux operating s...	4676	[OK]	
ubuntu-upstart	Upstart is an event-based replacement for ...	66	[OK]	
ubuntu-debootstrap	debootstrap --variant=minbase --components...	26	[OK]	

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
rastasheep/ubuntu-sshd	Dockerized SSH service, built on top of of...	40		[OK]
torusware/speedup-ubuntu	Always updated official Ubuntu docker imag...	27		[OK]

图 4.2 添加过滤条件搜索 Ubuntu 镜像

还有如--limit int 这样的参数用来限制显示数量，默认是返回 25 个结果。

4.1.34 管理服务的 docker service 命令

docker service 命令的作用是管理集群中的服务，需要与 docker swarm 配合使用。使用 docker service 时，主机必须是 swarm 的 manager。关于这部分的内容，会在第 18 章中详细介绍。

```
$ docker service --help

Usage: docker service COMMAND

Manage Docker services

Options:
  --help    Print usage

Commands:
  create    Create a new service
  inspect   Display detailed information on one or more services
  ps        List the tasks of a service
  ls        List services
  rm        Remove one or more services
  scale     Scale one or multiple services
  update    Update a service
```

Run 'docker service COMMAND --help' for more information on a command.

使用 docker service create 创建服务，使用 docker service ls 显示全部服务信息，使用 docker service ps 确认详细运行状况，docker create 命令提供了众多的可选参数，这些参数与其他流行的编排工具比较，基本上大同小异，使用方便。

如表 4.3 所示为 docker service 的命令说明。

表 4.3 Service 命令说明

命 令	说 明
service create	创建 service
service inspect	取得 service 的详细信息
service ps	取得 service 的任务信息
service ls	取得 service 列表信息
service rm	删除 service
service scale	调整 service 的 replicas
service update	更新 service

4.1.35 启动容器的 docker start 命令

docker start 命令的作用是启动一个或者多个停止状态的容器，docker start 命令后面可以是容器名称，也可以是容器 ID。

```
$ docker start --help
Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]
Start one or more stopped containers
Options:
```



```

-a, --attach          Attach STDOUT/STDERR and forward signals
--detach-keys string  Override the key sequence for detaching a
container
--help                Print usage
-i, --interactive     Attach container's STDIN

```

添加-a 参数选择显示 STDOUT / STDERR 信息；添加-i 参数显示 STDIN 信息。

4.1.36 查看容器状态的 docker stats 命令

docker stats 命令可以查看任何状态下的容器状态，一般用来查看运行时的容器对资源的使用情况。该命令输出的内容是实时更新的，取消查看时使用 **Ctrl+C** 键即可。

```

$ docker stats --help
Usage: docker stats [OPTIONS] [CONTAINER...]
Display a live stream of container(s) resource usage statistics
Options:
-a, --all          Show all containers (default shows just running)
--help            Print usage
--no-stream       Disable streaming stats and only pull the first result

```

如果不使用参数，直接使用 **docker stats** 命令会显示所有的正在运行的容器状态。如果要显示全部（包括非运行状态）的容器状态，可以添加-a 参数。

如果指定了某一个或者多个容器名称或者 ID，可以只显示指定容器的状态。

如果只想看某一时刻的状态，可以使用--no-stream 参数，这样终端会输出结束后自动返回可交互 shell 界面。

```

$ docker stats --no-stream nginx
CONTAINER CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS
nginx     0.00% 0 B / 0 B 0.00% 516.5 MB / 4.204 GB 230.4 kB / 0 B 5

```

4.1.37 停止容器的 docker stop 命令

docker stop 命令用于停止一个或者多个正在运行的容器，与 **docker kill** 命令不同，**docker stop** 命令会向容器发送正常停止的信号，而 **docker kill** 命令会强制终止容器进程，后者有可能造成数据丢失。

```

$ docker stop --help

Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop one or more running containers

Options:
--help      Print usage
-t, --time int Seconds to wait for stop before killing it (default 10)

```

通过添加-t 参数，可以在重启时设置等待容器停止的时间，如果容器在指定秒数之内没有停止，Docker 就会执行 **docker kill** 操作杀死容器，以便完成停止操作。

4.1.38 管理集群的 docker swarm 命令

docker swarm 命令与前面的 docker node、docker service 命令共同组成集群管理编排“三剑客”。

Docker 内置编排功能和目前成熟的编排工具相比可能还略显弱势，但毕竟它是 Docker 原生态的工具，占有得天独厚的优势。同时目前内置 swarm/node/service “三剑客” 所组成的组合，也能对普通的编排和应用场景提供足够的支持。

```
$ docker swarm --help
Usage: docker swarm COMMAND
Manage Docker Swarm
Options:
  --help    Print usage
Commands:
  init      Initialize a swarm
  join      Join a swarm as a node and/or manager
  join-token Manage join tokens
  update    Update the swarm
  leave     Leave a swarm
Run 'docker swarm COMMAND --help' for more information on a command.
```

为了更好地系统学习 Docker，这里不会过早对 Swarm 进行讲解，在第 18 章会具体讲解这几个命令。如表 4.4 所示为 docker swarm 的命令说明。

表 4.4 Swarm 命令说明

操 作	详 细 说 明
init	初始化集群
join	以 node (worker) 或者 manager 的身份加入集群
join-token	管理 join-token
update	更新集群
leave	退出集群

4.1.39 设置镜像标签的 docker tag 命令

docker tag 命令可以给镜像“重命名”，在 Docker 中镜像、容器、网络、数据卷等组件在运行时都会自动获取一个 ID，这个 ID 是独一无二的，但是这个 ID 实在太长了，不容易记忆，所以就有了 tag 的概念，给镜像打上标签，这样使用标签就可以找到响应镜像了。

```
$ docker tag --help

Usage: docker tag IMAGE[:TAG] IMAGE[:TAG]

Tag an image into a repository

Options:
  --help    Print usage
```

例如，给一个没有打标签的镜像打上标签：

```
$ docker tag a70c7fad1812 myusername/images:default
```

现在该镜像就可以使用 myusername/images:default 的名称了，还可以只修改镜像标签，

不改动名称:

```
$ docker tag myusername/images:default myusername/images:new
```

常用的情况还有修改名称以及标签:

```
$ docker tag myusername/images:default myusername/images-new:latest
```

4.1.40 查看容器进程的 docker top 命令

docker top 命令非常类似 Linux 下的 top 命令, 使用 docker top <Container ID/Name> 可以查看指定容器内正在运行的进程。

```
$ docker top --help
Usage: docker top CONTAINER [ps OPTIONS]
Display the running processes of a container
Options:
    --help    Print usage
```

例如, 查看一个 Nginx 容器的进程状态:

```
$ docker top nginx
UID          PID    PPID    C   STIME TTY   TIME         CMD
root         13149  13133   0   09:14 ?     00:00:00   nginx: master process nginx
-g daemon off;
systemd+    13191  13149   0   09:14 ?     00:00:05   nginx: worker process
systemd+    13192  13149   0   09:14 ?     00:00:03   nginx: worker process
systemd+    13193  13149   0   09:14 ?     00:00:06   nginx: worker process
systemd+    13194  13149   0   09:14 ?     00:00:12   nginx: worker process
```

4.1.41 恢复暂停容器的 docker unpause 命令

还记得前面的 docker pause 命令吗? 显然 docker unpause 是与其相对的命令, docker unpause 命令会恢复容器内的所有进程, 此时, 通过 docker stats 命令可以观察到资源使用情况是有变化的, 通过 docker logs -f 会输出日志。

docker unpause 命令会使 Cgroup 的 freezer 顺序恢复容器里的所有进程。

```
$ docker unpause --help
Usage: docker unpause CONTAINER [CONTAINER...]
Unpause all processes within one or more containers
Options:
    --help    Print usage
```

4.1.42 更新容器的 docker update 命令

对容器的设置更新时可以使用 docker update 命令, 使用时容器不需要停止或者重启。需要注意的是, 这里说的容器设置是指容器启动时添加的参数, 例如, 使用 docker run 命令时设置的 CPU 限制, 可以使用 docker update 命令取消。

docker update 命令不能使容器内部的应用程序重新读取程序的配置, 例如, 不能使一个 Nginx 容器重新加载配置, 如果需要重新加载 Nginx 则需要重启容器。

```
$ docker update --help
```

```
Usage: docker update CONTAINER [CONTAINER...]
```

Update configuration of one or more containers

```
--blkio-weight      Block IO (relative weight), between 10 and 1000
-c, --cpu-shares    CPU shares (relative weight)
--cpu-period        Limit CPU CFS (Completely Fair Scheduler) period
--cpu-quota         Limit CPU CFS (Completely Fair Scheduler) quota
--cpuset-cpus       CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems       MEMs in which to allow execution (0-3, 0,1)
--help             Print usage
--kernel-memory     Kernel memory limit
-m, --memory        Memory limit
--memory-reservation Memory soft limit
--memory-swap       Swap limit equal to memory plus swap: '-1' to enable
unlimited swap
--restart           Restart policy to apply when a container exits
```

在 `docker update` 命令的参数中, 大部分与 `docker run` 命令的参数相同, 唯一不同的是 `--restart` 参数, 添加该参数时不会立刻修改容器配置, 而是在未来容器重启时生效。

4.1.43 查看 Docker 版本的 `docker version` 命令

查看 Docker 版本, 使用 `-f` 参数可以格式化输出, 例如, `docker version -f'{{.Server}}'`。

```
$ docker version --help
```

```
Usage: docker version [OPTIONS]
```

Show the Docker version information

Options:

```
-f, --format string  Format the output using the given go template
--help             Print usage
```

```
$ docker version
```

Client:

```
Version:      1.12.1
API version:  1.24
Go version:   go1.6.3
Git commit:   23cf638
Built:       Thu Aug 18 05:02:53 2016
OS/Arch:     linux/amd64
```

Server:

```
Version:      1.12.1
API version:  1.24
Go version:   go1.6.3
Git commit:   23cf638
Built:       Thu Aug 18 05:02:53 2016
OS/Arch:     linux/amd64
```

4.1.44 管理数据卷的 `docker volume` 命令

`docker volume` 命令用来管理数据卷, 数据卷是容器数据持久化的一个组件, 数据卷是 Docker 体系第一部分, 在本书的第 9 章会单独介绍数据卷的使用与原理。

```
$ docker volume --help
```

Usage: docker volume COMMAND

Manage Docker volumes

Options:

--help Print usage

Commands:

create Create a volume

inspect Display detailed information on one or more volumes

ls List volumes

rm Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a command.

使用 `docker volume ls` 命令可以查看本机的全部数据卷, 因为数据卷与容器的生命周期完全独立, 所以容器删除后数据卷并不会删除, 时间长了数据卷就会越来越多。这个时候可以使用 `docker volume rm` 命令删除不需要的数据卷。使用 `docker volume inspect` 命令可以查看数据卷的详细信息, 使用 `docker volume create` 命令可以创建一个数据卷。

4.1.45 设置等待的 docker wait 命令

执行 `docker wait` 命令后, 该命令会“hang”在当前终端, 直到容器停止, 此时, 会打印出容器的退出码。该命令一般用在容器监控、异常捕捉方面。

```
$ docker wait --help
```

```
Usage: docker wait CONTAINER [CONTAINER...]
```

```
Block until a container stops, then print its exit code
```

```
Options:
```

```
--help Print usage https://docs.docker.com/engine/reference/commandline/
```

```
- annotations:S_U93nWfEea9u9OFqPi_gw#docker-management-commands
```

4.2 启动第一个 Docker 容器

4.1 节介绍了 45 个 Docker 命令之后, 相信读者已经对 Docker 有了整体认识, 下面通过一个简单的例子真正推开 Docker 的大门。

按照规律, 第一件事当然是打印一个 Hello World, 在 Docker 中同样存在这样的 Hello World, 下面直接运行 `hello-world` 镜像, 使用 `docker run` 命令时如果本地没有该镜像, 那么会自动从 Docker Hub 中拉取镜像。Docker Hub 是一个官方的镜像仓库。

```
// 这里使用的 --rm 表示容器退出后自动删除该容器
```

```
$ docker run --rm hello-world
```

```
Unable to find image 'hello-world:latest' locally
```

```
latest: Pulling from library/hello-world
```

```
c04b14da8d14: Pull complete
```

```
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1f
```

```
eb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent
    it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

上面加粗的内容表示是 hello-world 输出的内容，能显示这些信息则说明 Docker 运行正常了。接下来的章节会带领大家进入更详细地学习 Docker。

4.3 构建第一个 Docker 镜像

如果说 4.2 节是运行了一个 Hello World 镜像，那么现在我们来“写”一个自己的 Hello World 镜像。

新建一个文件夹 hello，然后在里面新建一个文件，名为 Dockerfile，注意大小写，然后写入下面两句：

```
FROM alpine
CMD "echo" "Hello World!"
```

第一句是表示即将构建的镜像基于一个叫做 Alpine 的镜像，在以后的学习中会发现绝大部分的镜像，都是基于某个基础镜像构建。基础镜像是指它们不是基于其他镜像构建而来的，它们往往是一个操作系统（发行版）精简之后构建而来，例如，Ubuntu、Debian、Alpine 等镜像，这些镜像一般托管在 Docker Hub 的 Library 仓库里。

保存文件，打开终端，下面使用 docker build 命令构建第一个镜像：

```
$ docker build -t hello .
```

-t 参数表示给构建的镜像打上名为 hello 的标签，构建完成后直接运行：

```
$ docker run --rm hello
Hello World
```

如此就算完成了一个镜像的构建与运行。

4.4 本章小结

从本章给出的两个简单的运行与构建例子中可以看到，Docker 面向用户的操作其实并不复杂，甚至非常友好，这也是 Docker 受欢迎的重要原因之一。本章介绍了 Docker 的 45 个命令，希望读者对 Docker 有一个整体了解，接下来的章节会逐步学习 Docker 的各个组件。

第 5 章 Docker 镜 像

虽然第 4 章讲了很多，但是还没有真正地操作 Docker 来做任何事情，只是输出了一个 Hello World。尽管如此，我们还是认识了 Docker 这种技术，知道了 Docker 是一种容器技术和 Docker 的运行原理，了解了 Docker 的历史，并且大致知道了 Docker 的子命令操作。

从本章开始将接触 Docker 最核心也是最基础的部分——镜像。Docker 镜像是 Docker 整个体系中最基础的一部分，Docker 镜像是容器的初始状态，Docker 镜像的构建和维护都对容器的运行有着极大的影响，所以理解镜像的原理对 Docker 后面的学习至关重要，在本章中将从最基础的拉取镜像到本地开始，逐步深入到镜像的存储原理和驱动等内容。

通过本章的学习，相信读者将完全掌握对镜像的基本操作，并对镜像的原理有一定了解。

5.1 认识镜像

获取镜像可以说是用户使用 Docker 的真正的第一步，在第 4 章中曾经介绍过了 docker pull 命令的基本使用方法，这条命令是用来获取已经构建并上传的镜像。

5.1.1 使用 docker pull 拉取镜像

Docker 提供了非常方便的拉取指令——docker pull，通过该命令可以拉取各个镜像仓库的镜像。要拉取一个镜像，最简单的格式就是 docker pull <imageName>，Docker 会自动从官方仓库搜寻匹配的镜像并开始拉取。一个标准并且完整的 pull 命令还应该包含标签，格式如下：

```
$ docker pull <imageName:tag>
```

对于 docker pull 命令，第 4 章中已经大致了解过，注意，如果没有指定标签，默认会拉取 latest 标签的镜像，一般情况下不推荐这样拉取镜像，因为在很多镜像标签中，latest 意味着不稳定，有可能这个不稳定的因素会影响用户的使用甚至威胁系统的安全，所以推荐使用带标签的 pull 操作。

对于私有仓库（私有仓库是一种用来托管因为各种原因不能放到官方仓库中的镜像仓库，在后面章节中会介绍到），还可以指定仓库地址：

```
$ docker pull localhost:5000/<imageName:tag>
```

5.1.2 搜索镜像

搜索镜像在第 4 章中已经讲得非常清楚，docker search 命令的使用也非常简单，但是除了在终端下搜索镜像外，还可以使用 Docker Hub 的网站搜索镜像，还有一些第三方的镜像仓库也拥有不少的镜像资源。

如图 5.1 所示为 Docker Hub 的首页，网址为 <https://hub.docker.com/>。

在以后的学习中还会频繁使用该网址，可以将其加入收藏夹中。



图 5.1 Docker Hub 首页

在右上角直接输入 Ubuntu 关键字，可以搜索相关的镜像，如图 5.2 所示，可以看到在网页版中搜索结果更加清晰方便（Docker Hub 的搜索引擎并不智能，用户需要正确输入想寻找的镜像名称才能返回相应的搜索结果）。

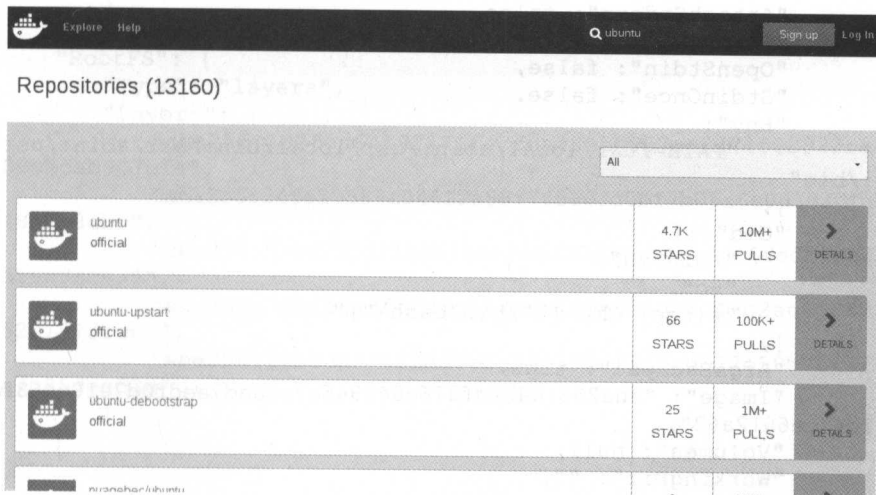


图 5.2 搜索 Ubuntu 镜像

一些国内的 Docker 初创公司也拥有自己的镜像仓库，用户还可以通过使用国内镜像仓库间接拉取官方仓库的镜像，以达到加速的效果，详细内容将在后面介绍。

5.1.3 查看镜像信息

最简单的查看镜像信息方法就是 `docker images` 命令，该命令可以查看本地存储的镜像，但是该命令只可以查看一些很简单的信息，如镜像名称、标签、镜像 ID、创建时间、大小等，更详细的命令则一般通过 `docker inspect` 命令来查看。

第4章中简单介绍过 `docker inspect` 的使用方法，`docker inspect` 命令不仅可以查看镜像信息还可以查看容器信息，直接使用 `docker inspect` 命令即可返回镜像的详细信息，下面通过查看 `Ubuntu:14.04` 镜像的详细信息来了解 `docker inspect` 指令。

```
$ docker inspect ubuntu:14.04
[
  {
    "Id": "sha256:4a725d3b3b1cc18c8cbd05358ffbbfedfeleb947f58061e5858f08e2899731ee",
    "RepoTags": [
      "ubuntu:14.04"
    ],
    "RepoDigests": [
      "ubuntu@sha256:5b5d48912298181c3c80086e7d3982029b288678fccabf2265899199c24d7f89"
    ],
    "Parent": "",
    "Comment": "",
    "Created": "2016-08-26T18:49:52.592709376Z",
    "Container": "c4f24a2f40edcb70bc67043d863d5d5f3ebda1db108c5a493c237f8276622f6b",
    "ContainerConfig": {
      "Hostname": "11687faae091",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
      ],
      "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) CMD [\"/bin/bash\"]"
      ],
      "ArgsEscaped": true,
      "Image": "sha256:04b2df876c043946c7cdaad7aedf0d7016cc68fa18019371e351d47ce6512a62",
      "Volumes": null,
      "WorkingDir": "",
      "Entrypoint": null,
      "OnBuild": null,
      "Labels": {}
    },
    "DockerVersion": "1.10.3",
    "Author": ""
  }
]
```

```

"Config": {
  "Hostname": "11687faae091",
  "Domainname": "",
  "User": "",
  "AttachStdin": false,
  "AttachStdout": false,
  "AttachStderr": false,
  "Tty": false,
  "OpenStdin": false,
  "StdinOnce": false,
  "Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin"
  ],
  "Cmd": [
    "/bin/bash"
  ],
  "ArgsEscaped": true,
  "Image": "sha256:04b2df876c043946c7cdaad7aedf0d7016cc68fa
18019371e351d47ce6512a62",
  "Volumes": null,
  "WorkingDir": "",
  "Entrypoint": null,
  "OnBuild": null,
  "Labels": {}
},
"Architecture": "amd64",
"Os": "linux",
"Size": 187935426,
"VirtualSize": 187935426,
"GraphDriver": {
  "Name": "devicemapper",
  "Data": {
    "DeviceId": "142",
    "DeviceName": "docker-8:1-1059258-525d5d36d4faec2009f4bd
60388cc668346c58b1c4e915c7731f630debd45b8e",
    "DeviceSize": "10737418240"
  }
},
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:102fca64f92471ff7fca48e55807ae2471502822ba620292
b0a06ebcab907cf4",
    "sha256:24fe29584c046f2a88f7f566dd0bf7b08a8c0d393dfad8370633
b0748bba8cbc",
    "sha256:530d731d21e1b1bbe356d70d3bca4d72d76fed89e90faab271d
29bd58c8ccea4",
    "sha256:344f56a35ff9fc747ada7d2b88bd21c49b2ec404872662cbaf
0a65201873c0c6",
    "sha256:ffb6ddc7582aa7e2e73f102df3ffcd272e59b7cf3f7abefe08
d11a7c85dea53a"
  ]
}
}
]

```

以上信息一般情况下不用这样输出，毕竟有些镜像的层非常多，输出信息量很大，所以结合 `docker inspect` 命令的参数可以更快地获得想要的信息，下面通过两个简单的例子来说明。

注意看上面的 **Created** 信息，显然要在那么多信息中查找是很麻烦的事情，所以通过 `-f` 参数就可以迅速定位，不仅方便查看，还方便开发第三方应用时获取信息：

```
$ docker inspect -f '镜像创建时间是: {{.Created}}' ubuntu:14.04
镜像创建时间是: 2016-08-26T18:49:52.592709376Z
```

这样就获得了详细的镜像创建日期。`-f` 的实参其实是个 Go 模板，并在容器、镜像的元数据上以该 Go 模板作为输入，最终返回模板指定的数据。Go 模板是一种模板引擎，让数据以指定的模式输出。这个概念对于 Web 开发者是非常熟悉的，Web 领域有很多模板引擎，如 **Jinga2**（用于 Python 和 Flask）、**Mustache**、**JSP** 等。具体的 Go 模板信息在 Golang 官网可以找到很详细的资料，这里不再赘述。

另一个常用的例子是把镜像信息输出为 JSON 格式：

```
$ docker inspect --format='{{json .Config}}' ubuntu:14.04
{"Hostname":"11687faae091","Domainname":"","User":"","AttachStdin":false,
"AttachStdout":false,"AttachStderr":false,"Tty":false,"OpenStdin":false,
"StdinOnce":false,"Env":["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"],"Cmd":["/bin/bash"],"ArgsEscaped":true,"Image":
"sha256:04b2df876c043946c7cdaad7aedef0d7016cc68fa18019371e351d47ce6512a62",
"Volumes":null,"WorkingDir":"","Entrypoint":null,"OnBuild":null,"Labels":{}}
```

这样的话，第三程序读取内容就会简单很多了。

5.2 创建镜像

镜像的创建可以说是整个 Docker 工作流程的第一步，有了镜像才能运行容器。创建一个 Docker 镜像常见的有两种方法：一种是通过 **Dockerfile** 构建，另一种是通过 `docker commit` 提交容器为镜像。

5.2.1 剖析 Hello World 镜像

在第 4 章中已经运行了 **hello-world** 镜像，这里不妨“打开”这个镜像，看看里面都是什么，如果已经将其删除了，可以使用 `docker pull` 命令拉回来，还记得 `pull` 操作就是拉取镜像吧，**hello-world** 镜像非常小。

```
$ docker pull hello-world
```

这个 Docker 镜像内部就只有一个 **hello** 二进制文件。构建时，镜像基于一个空的镜像，使用 **ADD** 把目录下的 **hello** 添加到镜像中，然后设置 **CMD** 启动命令，使镜像在运行时自动执行二进制文件。

5.2.2 从 Dockerfile 构建镜像

在后面的章节中会单独讲解 **Dockerfile** 的各种指令，在这里只是简单点到，创建镜像可以通过 **Dockerfile** 来构建，创建 **Dockerfile** 之前，先来看一下第 4 章运行过的 **hello-world** 镜像的 **Dockerfile**（Github 地址为 <https://github.com/docker-library/hello-world/tree/master/>）

hello-world)。

```
FROM scratch
COPY hello /
CMD ["/hello"]

然后使用 docker build 命令构建。

$ docker build -t hello
```

使用 docker ps 命令查看刚才构建的镜像，使用 docker run --rm hello 运行镜像，该镜像会输出 Hello World 信息。

5.2.3 自动构建镜像

在使用 Docker 镜像的过程中，我们经常需要构建自己的镜像，而每一次 docker build 与漫长的等待都非常耗费时间，而且面对一些大型镜像的编译工作还需要服务器有足够的硬件性能，这对普通用户来说是个不小的门槛与负担。

因此可以利用 Docker Hub 来自动构建镜像，解放我们的双手，也节省了一笔服务器费用。

在登录 Docker Hub 之后，首先在右上角头像的菜单中依次选择 Settings→Linked Accounts & Services，这时候可以看到 Github 的图标，单击 Link Github 按钮，然后 Docker Hub 就与 Github 仓库连接了。

接下来在右上角 Create 的下拉菜单中单击 Create Automated Build 即可打开自动构建的页面，如图 5.3 所示。

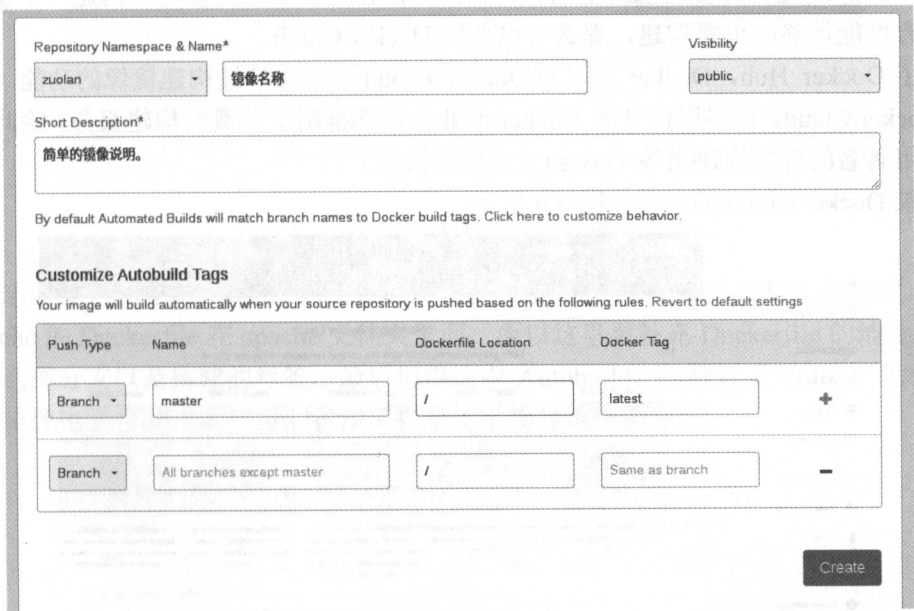


图 5.3 Docker Hub 自动构建界面

单击 Create 按钮之后，可以看到新的镜像页面已经搭建起来了，如图 5.4 所示，当 Github 上的 Dockerfile 仓库有改动时，Docker Hub 会自动构建镜像。

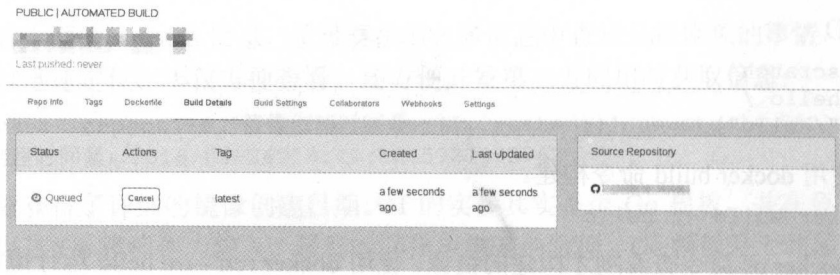


图 5.4 自动构建已经开始

构建完成会提示如下，用户可以单击相应的构建编号查看构建过程，如图 5.5 所示。

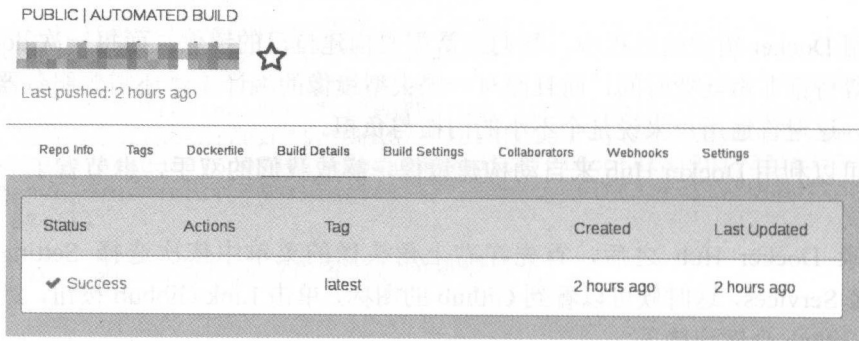


图 5.5 构建成功

Docker Hub 构建时会进入一个队列，并非立即构建，如果需要马上构建一个镜像，但手头没有性能足够的机器构建，那么可以使用 Docker Cloud。

除了 Docker Hub，Docker 公司的 Docker Cloud 也提供持续构建镜像的功能（当然，除了 Docker Cloud，很多国内外的 Docker 创业公司都推出了免费的构建服务，除此之外还可以使用著名的持续构建服务 Travis CI 来构建镜像）。

登录 Docker Cloud 的界面如图 5.6 所示。

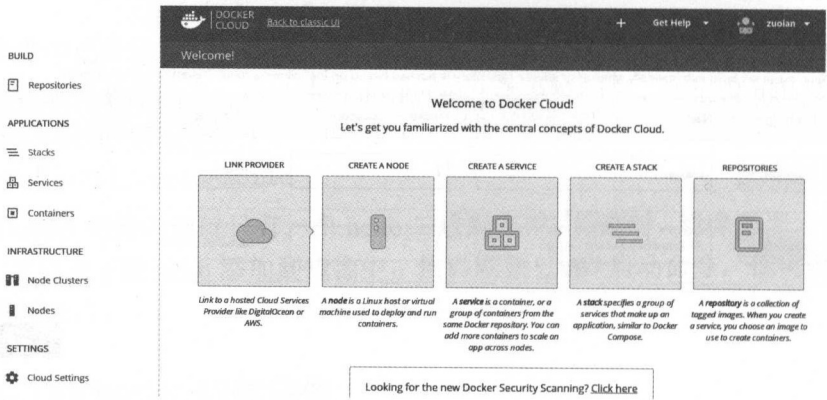


图 5.6 Docker Cloud 界面

选择左边的 Repositories 进入构建界面，如图 5.7 所示，这里以自动构建一个 Apache 镜像为例，选择之前构建的 Apache，准备设置为自动构建，选择连接到 Github。

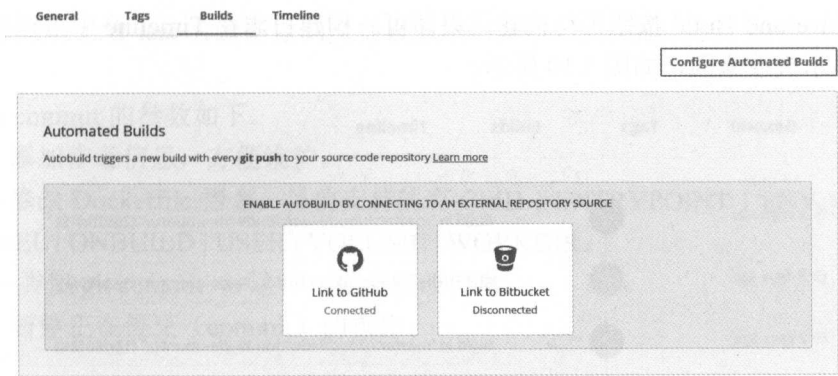


图 5.7 连接 Github 仓库

注意，目前构建功能处于 Beta 阶段，所以是免费的，以后应该会部分收费。其实除了 Docker Hub，其他很多公司都提供免费的构建服务，但为了更好地使用 Docker Hub，这里选择 Docker Cloud 作为例子。

如图 5.8 所示，选择 Dockerfile 仓库。

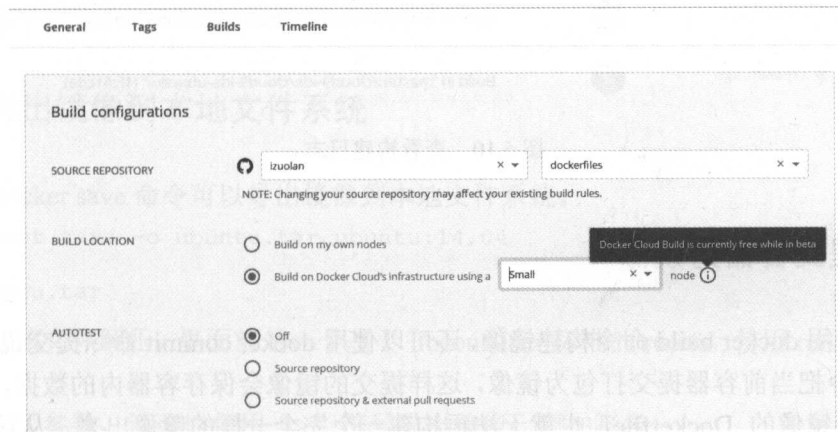


图 5.8 选择 Dockerfile 所在的 Github 仓库

注意图 5.9 中的 Dockerfile location 的填写，这里的/代表了仓库的根目录位置，而本例中 Apache 的 Dockerfile 在 apache 文件夹里面，所以这里要补充 Dockerfile 的地址/apache。还可以选择分支以及构建的标签，最后记得选中 Autobuild，这样以后 Github 仓库更新之后，该镜像也会自动重新构建，省却了自己动手构建推送的麻烦。

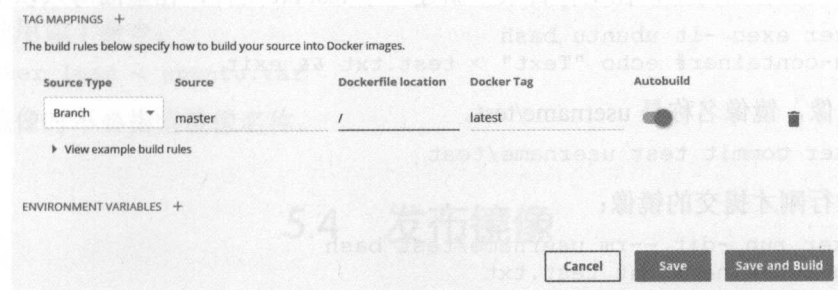


图 5.9 Dockerfile 位置

单击 **Save and Build** 按钮等待构建结果即可。构建日志在 **Timeline** 中可以查看，如构建失败会显示失败原因，如图 5.10 所示。

General	Tags	Builds	Timeline
🕒 2 days ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (26520d18)
🕒 2 days ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (5e3d03fa)
🕒 2 days ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (1bd861be)
🕒 22 days ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (9af19208)
🕒 24 days ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (b41523c1)
🕒 a month ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (4f60fc33)
🕒 a month ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (c848a5a9)
🕒 a month ago	✗		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (2d4d60f7)
🕒 2 months ago	✓		Build in 'master:/cloud9-ide/cloud9-ide-ubuntu/' (1fc61cda)

图 5.10 查看构建日志

5.2.4 提交容器为镜像

除了使用 `docker build` 命令构建镜像，还可以使用 `docker commit` 命令提交镜像。`docker commit` 命令把当前容器提交打包为镜像，这样提交的镜像会保存容器内的数据，而且第三方无法获得镜像的 `Dockerfile`，也就无法再构建一个完全一样的镜像出来，从这点看，并不推荐用户使用 `docker commit` 命令提交镜像。

但是有时需要使用 `docker commit` 命令来保存容器状态，因此这时还是需要使用这个方法保存容器的。下面以一个简单的例子进行说明。

首先启动一个容器。

```
$ docker run -d --name=test ubuntu:14.04
```

然后进入该容器内部，在工作目录下新建一个 `test.txt` 文件，在里面写入以下内容：

```
$ docker exec -it ubuntu bash
ubuntu-container# echo "Text" > test.txt && exit
```

提交镜像，镜像名称是 `username/test`。

```
$ docker commit test username/test
```

然后运行刚才提交的镜像：

```
$ docker run -dit --rm username/test bash
ubuntu-container# cat test.txt
ubuntu-container# Text
```

可以看到刚才的 `test` 容器新建的文件被保留下来了，`username/test` 镜像里面包含了该文件。

`docker commit` 的参数如下。

- `-a`: 添加作者信息，方便维护。
- `-c`: 修改 `Dockerfile` 指令，目前支持的有 `CMD` | `ENTRYPOINT` | `ENV` | `EXPOSE` | `LABEL` | `ONBUILD` | `USER` | `VOLUME` | `WORKDIR`。
- `-m`: 类似 `git commit -m` 这样，提交修改信息。
- `-p`: 暂停正在提交（commit）的操作。

5.3 导出和导入镜像

如果在两台主机之间需要传输镜像，办法是把镜像推送到仓库，然后让另一台主机拉回来，但是这样有个中转，不仅麻烦还不安全，有时候并不希望镜像发布到互联网中。而自己搭建私有镜像仓库显然不是三两个命令就能完成的，于是就需要一组可以导出和导入镜像的命令了。

5.3.1 导出镜像到本地文件系统

使用 `docker save` 命令可以导出镜像到本地文件系统。

```
$ docker save -o ubuntu.tar ubuntu:14.04
$ ls
$ ubuntu.tar
```

可以把该文件解压，里面就是一个基于 `libcontainer` 标准的 `rootfs`，使用 `runC` 也可以运行起来。

如果忘了参数，还可以使用“`>`”符号导出镜像，更加形象。

```
$ docker save ubuntu:14.04 > ubuntu.tar
```

5.3.2 从本地文件系统导入镜像

使用 `docker load` 命令可以加载一个导出的镜像包到本地仓库。

```
$ docker load -i ubuntu.tar
```

或者使用如下命令：

```
$ docker load < ubuntu.tar
```

导入镜像时不必指定镜像名称。

5.4 发布镜像

现在，我们已经知道了如何构建、导出、导入镜像，当想发布镜像让更多的人使用时，

就需要推送镜像到公共仓库了。

5.4.1 发布镜像到 Docker Hub

因为 Docker Hub 是官方默认仓库，镜像最多，因此一般都会选择发布到这里。读者也许还记得使用 `docker push` 命令可以推送镜像，但是在此之前，需要先在终端中登录到 Docker Hub。

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: username
Password:
Login Succeeded
```

这里输入密码时是不会显示输入反馈的，只需要输完密码后按 Enter 键就可以了。

登录成功后才可以使用 `docker push` 命令推送镜像：

```
$ docker push username/images
```

注意，这里推送的镜像名称表示镜像的所有者是 `username`，我们无法推送一个 `user/images` 的镜像到 Docker Hub，因为 `docker push` 命令只能推送镜像到用户有管理权限的仓库。这里的管理权限包括组织，与 Github 类似，Docker Hub 也有组织的概念。

5.4.2 给镜像打上标签

前面提到，我们只能推送镜像到自己有管理权限的仓库。假设现有用户名为 `username` 的用户，想推送一个 `user/image` 的镜像到 Docker Hub，有什么办法呢？

最简单的就是给镜像重新打上标签，重新打上标签之后，镜像内容不变，只是名称改变了。

```
$ docker tag user/image username/image
```

这样就可以推送 `username/image` 到 Docker Hub 了。

5.4.3 发布到第三方镜像仓库

虽然 Docker Hub 是官方默认的仓库，但是国内网络并不是很稳定，而且 Docker Hub 免费版本的功能有限制，因此国内许多 Docker 初创公司以及一些公益镜像仓库都开放了推送请求。用户可以选择把镜像推送到这些第三方仓库，不仅有助于加快推送速度，还方便以后拉取镜像时提速。

与前面类似，可以发现在使用 `docker push` 命令时会默认推送到 Docker Hub，为了改变这一默认值，还需要给镜像打上仓库标签。

```
$ docker tag username/image reg.example.com/username/image
```

这个时候使用 `docker images` 命令查看镜像会发现镜像名称已经改变，原来的标签不会删除。虽然变成两个镜像，但实际上只是占用一个镜像的空间。

如果要推送的仓库需要认证，别忘记使用 `docker login` 命令登录。


```
$ docker login reg.example.com
```

接下来就可以使用 `docker push` 命令推送了。

```
$ docker push reg.example.com/username/image
```

5.5 删除镜像

镜像多了，有些不需要的镜像就想要删掉。下面讲解如何删除镜像。

5.5.1 删除本地镜像

删除镜像的命令是 `docker rmi`，删除镜像时如不指定镜像的标签（tag）则会默认删除镜像的 `latest` 标签。可以在命令后面接上多个镜像名称，删除多个镜像。

使用 `docker rmi` 命令删除镜像时，要确保没有容器在使用该镜像，也就是没有容器是使用该镜像启动的，这样才可以删除，否则会报错。

删除镜像时可以使用镜像的 ID 也可以使用镜像名称，`docker rmi` 命令有一个参数 `-f`，该参数可以强制删除镜像，即便有容器正在使用该镜像。

但是这样只会删除镜像标签，不影响正在运行的容器，实际上只要容器还在运行，镜像就不会被真正删除，用户可以使用 `docker commit` 操作提交容器为镜像来恢复镜像。

```
// 删除一个镜像
$ docker rmi test
Untagged: test:latest
Deleted: sha256:c0ec52a519810bbab006186fe5ec107f477885601b13b29f0b1c940d03c2ac46
Deleted: sha256:f004c17c62d27346bd7ad32afd616d6f135ab7b7d67fa704906c3b6790133b59
// 删除一个标签
$ docker rmi volume_test
Untagged: volume_test:latest
```

前面说过，镜像实际上是以 ID 为标准保存在 Docker 中的，即使镜像没有使用标签，镜像也是可以存在的，出现这种情况的原因有很多，如构建时没有使用 `-t` 参数打上标签，或者强制删除了一个正在运行着容器的镜像，又或者构建的新镜像的 `tag` 覆盖了原来的旧镜像的 `tag` 等，时间长了，如果没有 `tag` 说明这些镜像是什么作用就会很难管理，因此需要删除这些 `<none>` 镜像，数量少时可以手动一条一条删除，数量多时可以配合 Docker 其他命令，删除所有未打 `dangling` 标签的镜像。

```
$ docker rmi $(docker images -q -f dangling=true)
```

删除所有镜像。

```
$ docker rmi $(docker images -q)
```

注意，shell 中的 `$()` 和 `"` 类似，会先执行这里面的内容。

5.5.2 删除仓库镜像

删除仓库镜像目前还没有 Docker 命令操作，不过可以通过 API 操作（可以参考本书

第16章)，一般第三方仓库都会提供可视化界面供用户删除镜像。

5.6 Docker 镜像扩展

前面对 Docker 镜像基础的介绍就是这么多，可以看到整个镜像的基础知识并不多，难点在 Dockerfile 的书写上，这一部分在后面会专门有一章来讲解，现在有个问题需要解决一下，那就是 Docker 镜像里面到底有什么？结构是怎样的呢？

5.6.1 Docker 镜像里有什么

在第2章的 runC 部分曾导出、解压过一个镜像，使用 docker save 命令导出一个镜像，然后解压该压缩包，得到文件如下：

```
$ ls
287820e08cab8daca28fe9543e43dfa33006d1e6ab30dc61dd38839e9e6243ca
b54d6e0d45851f9e27c3858595db82738ba91c93d59f918b2faf769ee300e0f3
3d54c79b787fb59fb927a8e275610f62e7c2fd3543d9d9372133bc6c12f166e3
d189d5ffa6772e6b8b2b57f845c6c1b6933761608b0b07311664fc5bc7658a99.json
40c500c3041e33a21d6067a6313861d2dc505c401d3b35b452b97b1b50b4d58d
f3912b6e71abfeaf1553e47fa73e203abfd60fc29b67fedd11dfebdc94e52bbf
7e111f7d7f6f835b853fba7500417db78eea330c7a0629df44b1576fecb9ae30
manifest.json
9e60893396bbf0d8f82d124c3e996a44ff28a3031eb12a7bcecf031450789edc
repositories
a7276a48da6650a6aaf30bd9b566248f82662b8187b4e99ec511d7610813586f
```

这些像乱码一样的文件夹其实是镜像的一个层（layer），前面提到过，镜像包含着数据以及必要的元数据，这些数据就是层（layer），而元数据则是一些 JSON 文件，元数据是用来描述镜像的信息，包括数据之间的关系、容器配置信息等。

上面解压的镜像所显示的每一个层（layer）文件夹意味着它是由一句 Dockerfile 指令生成的。在构建镜像过程中，像 RUN、COPY、ADD、CMD 等指令都会生成一个新的镜像层，一个镜像就是不断在上一个镜像层的基础上叠加上去的。

为了更直观地了解一个镜像的历史，可以使用 docker history 命令来查看镜像历史（为了显示得好看些，输出内容已经做了整理）：

```
$ docker history nginx:alpine
IMAGE          CREATED          CREATED BY                                      SIZE
935bd7bf8ea68  8 weeks ago     /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon o  0 B
<missing>      8 weeks ago     /bin/sh -c #(nop) EXPOSE 443/tcp 80/tcp          0 B
<missing>      8 weeks ago     /bin/sh -c #(nop) COPY file:d15ceb73c6ea776c2  1.097 kB
<missing>      8 weeks ago     /bin/sh -c #(nop) COPY file:af94db45bb7e4b8ff  643 B
<missing>      8 weeks ago     /bin/sh -c GPG_KEYS=B0F4253373F8F6F510D421785  50.01 MB
<missing>      8 weeks ago     /bin/sh -c #(nop) ENV NGINX_VERSION=1.11.3      0 B
<missing>      8 weeks ago     /bin/sh -c #(nop) MAINTAINER NGINX Docker Mai  0 B
<missing>      3 months ago    /bin/sh -c #(nop) ADD file:852e9d0cb9d906535a  4.795 MB
```

可以看到 Nginx 的镜像由 8 句指令构建而成，构建过程由下到上，一层一层叠加，最开始是 ADD 一个 Alpine 镜像作为基础镜像，然后是 MAINTAINER 信息，再然后是 ENV 设置，继续往上依次是构建过程中的指令，直到最后一句 CMD 指令结束镜像构建，这样

就获得了一个基于 Alpine 的 Nginx 镜像。

知道了镜像内部是什么样之后，在本地 Docker 是如何存储这些镜像的呢？那么就去 Docker 本地存储路径一探究竟吧。

```
$ sudo ls -l /var/lib/docker
总用量 40
drwx----- 17 root root 4096 10月 20 23:54 containers // 存放容器的信息
drwx----- 5 root root 4096 8月 20 16:46 devicemapper
drwx----- 3 root root 4096 7月 19 16:34 image // 存放镜像的信息
drwxr-x--- 3 root root 4096 7月 19 16:34 network // 存放网络信息
drwx----- 2 root root 4096 7月 29 22:08 swarm // 存放集群信息
drwx----- 2 root root 4096 10月 21 19:25 tmp
drwx----- 2 root root 4096 7月 19 16:34 trust
drwx----- 64 root root 12288 10月 20 09:08 volumes // 存放数据卷信息
```

本地存储的镜像数据与层数据在 image 文件夹中是分开存储的，imagedb 保存了本地全部镜像的元数据，而 layerdb 文件夹保存了本地镜像的全部镜像层。

5.6.2 Docker 镜像的存储方式

前面说到镜像内容与元数据是分开存储的，那么 Docker 是如何把这些内容整合然后把一个完整镜像显示在用户眼前的呢？

以 Nginx 镜像为例，使用 `docker inspect` 命令查看镜像信息（只截取 `rootfs` 部分）：

```
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:4fe15f8d0ae69e169824f25f1d4da3015a48feeeebbb265cd2e328e15c6a869f",
    "sha256:da85d9b3377006fe88628078bd0b0418b81fc9d4018f903169a716214479ee64",
    "sha256:b36519590516cd2cfda9f00c20ce06d64f0375e1ad895cd26827a2dbdb9e2e1d",
    "sha256:1bbf5c4f940bae3e55efe6bbf4433361f5101727f7440755f2c4e87a6b967297"
  ]
}
```

可以看到 Docker daemon 首先通过 image 的元数据得知全部 layer 的 ID，再根据 layer 的元数据梳理出顺序，最后使用联合挂载技术还原容器启动所需要的 rootfs 和基本配置信息。运行的容器实际上就像是在这些镜像层之上新建一个动态的层。

5.6.3 联合挂载

前面多次提到了一个名词：联合挂载。联合挂载会把多个目录挂载到同一个目录（甚至可能对应不同的文件系统）下，并对外显示这些目录的整合形态。Docker 中使用的 AUFS (AnotherUnionFS) 就是一种联合文件系统。

联合文件系统在日常计算机中有一个地方会经常用到，那就是 Linux 系统的 LiveCD，我们使用发行版时一般都有一个 LiveCD 供用户体验。其原理就是在原有的系统目录之上附加一层可读可写的文件层，任何文件改动都会被写入这个文件层中，这种技术就是写时

复制。关于写时复制的信息可以查看 Overlay 文件系统的资料。

因为写时复制的特点，需要非常注意的一点就是在以后构建镜像过程中，大部分用户都会有一个误区就是删除一个文件必定会导致镜像体积变小。

实际上并非如此，举个简单的例子，现在有一个镜像，内部有一个 100 MB 的文件，现在基于该镜像（FROM 指令）构建一个新的镜像，在构建过程中执行了删除那个 100 MB 的文件的命令，那么现在镜像体积变小了吗？

当然没有，因为根据联合挂载与写时复制的特点，删除底层文件系统的文件或者目录时，会在上层建立一个同名的主设备号都为 0 的字符设备，并不会删除底层文件系统的文件或者目录，只是整合后的 rootfs 让用户看不到那些文件而已。

所以正确的解决办法是从底层的文件系统着手，在最初的镜像层删掉 100 MB 文件才是减少镜像体积的办法。更详细的内容会在以后实战过程中提示。

5.6.4 Git 式管理

Git 估计大家都用过，这是一种管理代码的分布式版本控制工具。Docker 作为一个近年来最具开创性的项目，充分借鉴了 Git 的优点来管理镜像，我们之前学习过的 pull、push、commit 的命令都很像 Git 的操作。Docker 通过把镜像分层使得层的复用成为可能，如两个镜像使用有几个镜像层是一样的，那么它们可以公用这些镜像层。

除此之外，Docker Hub 也类似 Github 一样，可以说变革了容器应用的分发流程。

5.7 本章小结

本章主要介绍了 Docker 镜像的使用与原理，包括拉取、构建、推送、删除、导出和导入等。在以后的使用过程中，还会针对实战例子做出一些补充说明。

可以说镜像是 Docker 的基础，在以后的学习过程中会有不少需要自己动手构建的镜像，读者不妨注意积累管理自己的 Dockerfile。

第 6 章会详细介绍 Dockerfile 的写法，熟练掌握镜像构建的方法。

第 6 章 Dockerfile 文件

第 5 章中已经认识了 Docker 镜像, 那么对于构建 Docker 镜像的“圣旨”——Dockerfile 将在本章继续深入学习。

前面已经很多次提到可以使用 Dockerfile 构建镜像或者使用 `docker commit` 提交镜像。这两种方法生成的镜像最大的区别在于: 前者可以通过一份简单的文本就把整个镜像概括进去, 其他人只需要拿到 Dockerfile 就可以构建出一个“一模一样”的镜像; 而后者(使用 `docker commit`)生成的镜像, 其他人只能通过 Registry 或者导出导入的方式来传输镜像, 非常不方便, 而且其他人很难确定镜像内有什么, 也无法构建一个“一模一样”的镜像出来。所以一般不推荐使用 `docker commit` 的方式生成镜像。

因此掌握使用 Dockerfile 构建镜像就很有必要了。Dockerfile 的内容并不多, 配合 `docker build` 命令的使用, 可以轻松构建一个自己定制的镜像。

本章将逐一解释 Dockerfile 的每条指令, 然后再解释镜像构建过程中的扩展功能, 如 `.dockerignore` 等。

本章主要内容有:

- Dockerfile 基本结构;
- Dockerfile 指令;
- 使用 Dockerfile 构建镜像。

6.1 Dockerfile 基本结构

Dockerfile 就像一道“圣旨”, 任何人拿着这份文件执行 `docker build` 命令都会生成相同功能的镜像。而为了让这样一份文本有更高的可读性及通用性, 文本结构就很讲究了, 就像写书信一样, 有着一定的格式。

6.1.1 Dockerfile 基础

要写一份 Dockerfile, 需要作者有一定的 Linux 命令行基础。因为整体来看, Dockerfile 就是一份自动化的 Linux 命令集。

在写 Dockerfile 过程中, 需要模拟一遍命令的运行过程, 尽量减少因命令行写错而重新构建的情况, 因为这很耗时。

命名规则:

Dockerfile 这个文件虽然可以命名为其他名字, 但是一般情况下不推荐修改。Dockerfile 这个文件名, 除非同一个文件夹下存在多个 Dockerfile 文件, 此时可以使用

Dockerfile.second、Dockerfile.server 等方式命名，构建时加上-f 指定该文件即可。

之所以不建议使用其他命名，是因为 Dockerfile 本身只是一份文本，不带特殊权限，改为其他名字后，会导致作者以外的人仅看文件名不容易分辨。部分自动化构建工具甚至默认使用 Dockerfile 这个名字。

6.1.2 Dockerfile 的书写规则

默认的 Dockerfile 文件名字就叫 Dockerfile，不带后缀，首字母大写。一份标准的 Dockerfile 中应包含指令、注释等内容，构建指令应使镜像尽量干净，不留垃圾文件。

Dockerfile 的结构如下：

```
// 这是注释
// 一般来说结构形如“指令 参数”，虽然官方允许使用小写的指令，但为了更好阅读
Dockerfile，一般把指令全部大写
INSTRUCTION arguments
```

Dockerfile 书写思想：

所谓书写思想，就是指怎样写一份 Dockerfile 才是正确的 Dockerfile，构建出符合 Docker 理念的镜像。

首先就是“自动化”。因为 Docker 构建过程是无交互的，所以整个构建过程需要保证命令集能够一直持续不断地执行下去。因此在书写 Dockerfile 过程中，需要注意命令是否能够自动执行，遇到交互节点是否可以自动应答等。（在安装依赖时不要执行软件升级操作，如 apt-get upgrade 等行为都是破坏镜像兼容性的做法，可能会导致其他人构建时产生因为版本问题而构建失败的问题。）

第二点是顺序。因为 Dockerfile 构建过程是从上到下，所以书写 Dockerfile 时需要考虑到后面的命令执行情况，并适当调整指令位置。例如，在执行删除软件源命令之前一般要完成全部软件依赖的安装，不能在发现缺少依赖时再重新添加软件源进行安装。（如果调整 Dockerfile 上面的指令，后面指令即使没有改动，构建时也会从改动处开始构建。）

第三点是清理。这是很多人在书写 Dockerfile 时没有注意到的地方。有时一个镜像在构建过程中会产生很多临时文件，很多时候在完成构建之后，临时文件也保留在了镜像之中。因此在 Dockerfile 中一般在最后会写上清理系统的命令，以保证镜像体积。值得注意的是，并不是删除文件后镜像体积一定会减小，这要根据 Docker 的存储特点和 Dockerfile 文件来判断。

第四点就是易读。有时候使用一条很长的命令，不要一行写到底，这样不容易阅读，遇到长命令可以使用“\”符号来连接，遇到几个命令连在一起，还可以使用“\&&”的方式连接。

```
RUN echo 'we are running some things' \
&& echo 'Hello'
```

更规范的写法可以参考官方的文档 https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/。

6.1.3 基础镜像信息和维护者信息

一般情况下可以在 Dockerfile 顶部使用注释的方式补充一些关于镜像的信息，可以包括用途、用法、版本及维护者的信息，这将有利于他人从中获得更全面的帮助。

6.2 Dockerfile 指令

6.2.1 指定基础镜像的 FROM 指令

FROM 指令表示将来构建的镜像是来自哪个镜像，也就是使用哪个镜像作为基础进行构建的。一般情况下 Dockerfile 都有基础镜像，FROM 指令必须是整个 Dockerfile 第一句有效指令。

FROM 的格式如下：

```
FROM <imageName:tag>
```

当同一个 Dockerfile 构建几个镜像时，可以写多个 FROM 指令，比如同时用 Ubuntu 和 Debian 作为基础镜像构建一个系列的镜像，执行构建会使用最后一句 FROM 语句。

6.2.2 设置维护者信息的 MAINTAINER 指令

MAINTAINER 指令格式如下：

```
MAINTAINER Name <Email>
```

这条指令主要是指定维护者信息，方便他人寻找作者。指令后面的内容其实没有规定写什么，只要可以联系上作者即可，一般使用邮箱地址。

6.2.3 执行构建命令的 RUN 指令

接下来是 RUN 指令。这条指令用来在 Docker 的编译环境中运行指定命令。RUN 会在 shell 或者 exec 的环境下执行命令。

shell 格式如下：

```
RUN echo HelloWorld
```

RUN 指令会在当前镜像的顶层执行任何命令，并 commit 成新的（中间）镜像，提交的镜像会在后面继续用到。

RUN 指令还有另外一种格式（exec 格式）：

```
RUN ["程序名", "参数 1", "参数 2"]
```

这种格式运行程序，可以免除运行/bin/sh 的消耗。这种格式使用 JSON 格式将程序名与所需参数组成一个字符串数组，所以如果参数中有引号等特殊字符，需要进行转义。

exec 格式不会触发 shell, 所以\$HOME 这样的环境变量无法使用, 但它可以在没有 bash 的镜像中执行, 而且可以避免错误的解析命令字符串。

6.2.4 设置镜像环境变量的 ENV 指令

ENV 指令用来指定在执行 docker run 命令运行镜像时, 自动设置的环境变量。这个环境变量可以在后续任何 RUN 指令中使用, 并在容器运行时保持, 而且可以通过 docker run 命令的-e 参数来修改环境变量。

ENV 指令语法如下:

```
ENV <key> <value>
```

使用 ENV 指令类似于 Linux 下的 export 命令, 用户可以在后续 Dockerfile 中使用这个变量。例如:

```
ENV TARGET_DIR /app
WORKDIR $TARGET_DIR
```

6.2.5 复制文件的 COPY 指令

COPY 指令用来将本地的文件或文件夹复制到镜像的指定路径下。格式如下:

```
COPY /Local/Path/File /Images/Path/File
```


6.2.6 添加文件的 ADD 指令

ADD 和 COPY 作用相似, 但实现不同。ADD 指令可以从一个 URL 地址下载内容复制到容器的文件系统中, 还可以将压缩打包格式的文件解开后复制到指定位置。

ADD 指令格式如下:

```
ADD File /Images/Path/File
ADD latest.tar.gz /var/www/
```

在相同的复制命令下, 使用 ADD 构建镜像的大小比 COPY 指令构建的镜像要大, 所以如果只是复制文件请使用 COPY 指令。

 注意:

- 不能对构建目录或上下文之外的文件进行 ADD 操作, 即不能使用 ../path 这样的路径。
- 如果目的位置不存在, 会自动创建。
- ADD 指令会使得构建缓存无效。

6.2.7 指定端口暴露的 EXPOSE 指令

EXPOSE 指令格式如下:

```
EXPOSE <端口> [<端口>...]
```

EXPOSE 指令用于标明这个镜像中的应用将会侦听某个端口，并且希望能将这个端口映射到主机的网络界面上。但是为了安全，docker run 命令如果没有带上响应的端口映射参数，Docker 并不会将端口映射出去。

此外 EXPOSE 端口是可以在多个容器之间通信用 (links)，通过--links 参数可以让多个容器通过端口连接在一起。

6.2.8 设置镜像启动命令的 CMD 指令

CMD 提供了容器默认的执行命令。Dockerfile 只允许使用一次 CMD 指令。使用多个 CMD 会抵消之前所有的指令，只有最后一个指令生效。一般来说这是整个 Dockerfile 脚本的最后一条指令。当 Dockerfile 已经完成了所有环境的安装与配置，通过 CMD 指令来指示 docker run 命令运行镜像时要执行的命令。格式如下：

```
CMD ["executable","param1","param2"]
```

值得注意的是，docker run 命令可以覆盖 CMD 命令。CMD 与 ENTRYPOINT 的功能极为相似。区别在于：如果 docker run 后面出现与 CMD 指定的相同的命令，那么 CMD 会被覆盖；而 ENTRYPOINT 会把容器名后面的所有内容都当成参数传递给它指定的命令（不会对命令覆盖）。另外，CMD 指令还可以单独作为 ENTRYPOINT 指令的可选参数，共同组合成一条完整的启动命令。

下面以例子说明，实验 Dockerfile 如下：

```
FROM ubuntu
CMD ["echo", "Hello Ubuntu"]
```

然后构建镜像并运行容器，运行时会返回：

```
$ docker build -t test .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu
---> bd3d4369aebc
Step 2 : CMD echo 'Hello Ubuntu'
---> Running in 14c9aa5280a9
---> a8391a058561
Removing intermediate container 14c9aa5280a9
Successfully built a8391a058561
$ docker run test
Hello Ubuntu
$ docker run test echo "Hello Docker"
Hello Docker
```

当使用 docker run test echo 方式启动容器时，echo "Hello Docker" 命令会覆盖原有 CMD 指令。也就是说 CMD 指令可以通过 docker run 命令覆盖。这一点也是 CMD 和 ENTRYPOINT 指令的最大区别。

CMD 与 RUN 的区别在于，RUN 是在 build 成镜像时就运行的，先于 CMD 和 ENTRYPOINT，CMD 会在每次启动容器的时候运行，而 RUN 只在创建镜像时执行一次，固化在 image 中。

6.2.9 设置接入点的 ENTRYPOINT 指令

前面已经说到了 ENTRYPOINT 指令。这个指令和 CMD 很相似。ENTRYPOINT 相当于把镜像变成一个固定的命令工具，它一般是不可以通过 docker run 来改变的。而 CMD 不同，CMD 是可以通过启动命令修改内容的。二者的主要区别通过实践会体会得更清晰。实验 Dockerfile 如下：

```
FROM ubuntu
ENTRYPOINT ["echo"]
```

实验过程如下：

```
$ docker build -t test .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu
----> bd3d4369aebc
Step 2 : ENTRYPOINT echo
----> Running in c6b9b6a657fd
----> af1c6cd7f531
Removing intermediate container c6b9b6a657fd
Successfully built af1c6cd7f531
$ docker run test "Hello Docker"
Hello Docker
```

可以看到，在 ENTRYPOINT 指令下，容器就像一个 echo 程序，docker run 后续参数就成了 echo 的参数。

6.2.10 设置数据卷的 VOLUME 指令

VOLUME 指令用来向基于镜像创建的容器添加数据卷（在容器中设置一个挂载点，可以用来让其他容器挂载或让宿主机访问，以实现数据共享或对容器数据的备份、恢复或迁移）。数据卷可以在容器间共享和重用。数据卷的修改是立即生效的。数据卷的修改不会对更新镜像产生影响。数据卷会一直存在，直到没有任何容器使用它（没有使用它也会在宿主机存在，但就不是数据卷了，和普通文件无异）。VOLUME 指令在后面还会详细介绍，这里只做简单使用说明。

VOLUME 指令格式如下：

```
VOLUME ["/data","/data2"]
VOLUME /data
```

VOLUME 可以在 docker run 中使用。如果 run 命令中没有使用，则默认不会在宿主机挂载这个数据卷。如果 Dockerfile 中没有设置数据卷，在 docker run 中也是可以设置的。在 Dockerfile 中声明数据卷有助于开发人员迅速定位需要保存数据的目录位置。

下面用实例说明。写一份 Dockerfile 如下：

```
FROM ubuntu
RUN mkdir /app && echo "Hello" > /app/test.txt
VOLUME /home
CMD ["cat", "/app/test.txt"]
```

然后构建：

```

$ docker build -t test .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu
---> bd3d4369aebc
Step 2 : RUN mkdir /app && echo "Hello" > /app/test.txt
---> Running in delc060fec4c
---> cef195e37ae4
Removing intermediate container delc060fec4c
Step 3 : VOLUME /home
---> Running in 03256c21c57c
---> 2625346dd697
Removing intermediate container 03256c21c57c
Step 4 : CMD cat /app/test.txt
---> Running in daac6eaffe2f
---> 75254f6ac08d
Removing intermediate container daac6eaffe2f
Successfully built 75254f6ac08d

```

直接运行容器，看到返回 Hello 的信息，说明 cat 命令的目标文件是容器内部的 /app/test.txt 文件。

```

$ docker run --rm test
Hello

```

在本地创建一个文件：

```

$ mkdir local
$ echo "Here is test" > ~/local/test.txt

```

再运行容器，这时设置了一个数据卷。注意，上面的 Dockerfile 并没有设置 /app 为数据卷，但是在 docker run 中使用了 -v 参数指定了 /app 目录，查看 cat 的结果会发现目标文件并不是容器内部的 test.txt 文件，而是宿主机上的 test.txt 文件。

```

$ docker run --rm -v ~/local:/app test
Here is not test

```

6.2.11 设置构建用户的 USER 指令

USER 指令格式如下：

```

USER user
USER user:group
USER uid:gid

```

USER 指定运行容器时的用户名或 UID（默认为 root）。后续的 RUN 也会使用指定用户。

USER 指令可以在 docker run 命令中通过 -u 选项来覆盖。这条指令应用场景在于，当服务不需要管理员权限时可以通过该命令指定运行用户。指定的用户需要在 USER 指令之前创建。例如：

```

RUN groupadd -r newuser && useradd -r -g newuser newuser

```

要临时获取管理员权限可以使用 `gosu`，而不推荐 `sudo`。

6.2.12 设置工作目录的 WORKDIR 指令

WORKDIR 指令指定 RUN、CMD 与 ENTRYPOINT 命令的工作目录。语法如下：

```
WORKDIR /path/to/workdir
```

同样，`docker run` 可以通过 `-w` 标志在运行时覆盖指令指定的目录。

此外，可以使用多个 `WORKDIR` 指令，后续命令的参数如果是相对路径，则会基于之前命令指定的路径。例如：

```
WORKDIR /a
WORKDIR b
WORKDIR c
```

则最终路径为 `/a/b/c`。

6.2.13 设置二次构建指令的 ONBUILD 指令

`ONBUILD` 指定在构建镜像时并不执行，而是在它的子镜像中执行。下面以一个简单的例子来说明。

Dockerfile 文件如下：

```
FROM busybox
ONBUILD RUN echo "You won't see me until later"
```

构建如下：

```
$ docker build -t me/no_echo_here .
Uploading context 2.56 kB
Uploading context
Step 0 : FROM busybox
Pulling repository busybox
769b9341d937: Download complete
511136ea3c5a: Download complete
bf747efa0e2f: Download complete
48e5f45168b9: Download complete
---&gt; 769b9341d937
Step 1 : ONBUILD RUN echo "You won't see me until later"
---&gt; Running in 6bf1e8f65f00
---&gt; f864c417cc99
Successfully built f864c417cc9
```

可以看到，在构建过程中会读取 `ONBUILD` 的指令，但是并不会执行。接下来用上面的镜像来构建子镜像。

Dockerfile 文件如下：

```
FROM me/no_echo_here
```

构建如下：

```
$ docker build -t me/echo_here .
Uploading context 2.56 kB
Uploading context
Step 0 : FROM cpuguy83/no_echo_here

# Executing 1 build triggers
Step onbuild-0 : RUN echo "You won't see me until later"
---&gt; Running in ebfe7e39c8
You won't see me until later
---&gt; ca6f025712d4
---&gt; ca6f025712d4
Successfully built ca6f025712d4
```


可以看到，在这一次构建中 ONBUILD 的指令被执行了。目前 ONBUILD 指令后面不能是 FROM 和 MAINTAINER 指令，当然也不能是 ONBUILD 自己。类似于 ONBUILD FROM 的都是错误的指令。

6.2.14 设置元数据的 LABEL 指令

LABEL 指令添加元数据到镜像。每一个标签会生成一个 layer，所以尽量使用一个 LABEL 标签。例如：

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

或者：

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

标签信息会保存到镜像中，如果有某个值已经存在，新的标签元素会覆盖它。

6.2.15 设置构建变量的 ARG 指令

ARG 指令定义了一个变量，用户可以在构建时使用，效果和 `docker build --build-arg <varname>=<value>` 一样，可以在构建时设定参数，这个参数只会在构建时存在。

与 ENV 类似，不同的是 ENV 会在镜像构建结束之后依旧存在镜像中，而 ARG 会在镜像构建结束之后消失。

6.2.16 设置停止信号的 STOPSIGNAL 指令

STOPSIGNAL 指令允许用户定制化运行 `docker stop` 时的信号。例如：

```
STOPSIGNAL SIGKILL
```

基于这种方式构建的镜像，其启动的容器在停止时会发送 SIGKILL 信号。这个指令适用于一些不能接受正常退出信号的容器。

6.2.17 检查镜像状态的 HEALTHCHECK 指令

这是一个健康检查指令，用来检查将来容器启动运行时是否正常。正常则会返回 healthy，否则返回 unhealthy。格式如下：

```
HEALTHCHECK [OPTIONS] CMD command
```

参数有 3 个：

(1) 设置在容器启动多长时间后开始检查容器状态如下。

```
--interval=DURATION (默认 30s)
```

(2) 设置超时时间，超过这个时间不返回信息，表示容器异常如下。

```
--timeout=DURATION (默认 30s)
```

(3) 设置重试次数如下。

```
--retries=N (默认 3)
```

例如：

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

这样可以在构建时检查容器的 CMD 指令是否运行正常，不需要在构建结束后再运行容器来测试。

6.2.18 设置命令执行环境的 SHELL 指令

在 Docker 构建过程中，默认会使用/bin/sh 作为 shell 环境。Windows 下构建默认使用 cmd 作为 shell 环境。但是有时需要在其他 shell 环境中执行 RUN 的内容，这时就需要用 SHELL 指令提醒 Docker 更换 shell 环境。

例如，在 Windows 下更换 powershell 为默认 shell 环境。

```
SHELL ["powershell", "-command"]
```

6.3 镜像构建实战

至此已经学完了 Docker 镜像的基础知识。接下来用一个稍微复杂的实战例子，来回顾之前所学的内容。

6.3.1 收集应用信息

本节实战内容是构建 pagekit CMS 镜像。这是一个使用 MIT 协议开源的 PHP 内容管理系统（CMS）。之所以选择这个应用作为例子，是考虑到 pagekit 本身并不复杂，可以使用 Sqlite 作为数据库，从而避免出现复杂的容器通信，同时 pagekit 需要的 PHP 环境是大家都比较熟悉的，而且 pagekit 的目录结构清晰，所以有助于设置数据卷，保持数据持久化。如图 6.1 所示为 pagekit 的 Logo。

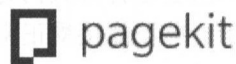


图 6.1 Pagekit Logo

首先，pagekit 是一个 PHP 程序，必然需要 PHP 环境（PHP 5）。其次需要一个 Web 服务器（Nginx）。然后是 pagekit 运行时需要的 PHP 扩展（php5-fpm php5-cli php5-json php5-mysql php5-curl）。除了这些，还需要基本的服务组件（ca-certificates）。

基本上一个 pagekit 镜像就需要这几个软件了。在 Github 上可以查看 pagekit 的说明，网址为 <https://github.com/pagekit/pagekit>。

6.3.2 编写 Dockerfile

有了前面的信息收集，现在可以写 Dockerfile 了。首先镜像基于 Ubuntu，再填写镜像维护者信息。

```
FROM ubuntu:trusty
MAINTAINER UserName<i@example.com>
```

然后安装运行环境:

```
RUN apt-get update && \
    apt-get -y install \
    nginx \
    unzip \
    wget \
    ca-certificates \
    php5 php5-fpm php5-cli php5-json php5-mysql php5-curl
```

使用“\”符号可以换行显示,便于阅读。前面安装了 pagekit 运行时所需要的环境以及后面需要用的“临时软件”wget。这个软件并不是 pagekit 运行所必需的,但是之后的构建需要用到它,因此需要安装它,后面还会卸载它。接下来开始安装 pagekit。

```
ENV PAGEKIT_VERSION 1.0.2
RUN mkdir /pagekit
WORKDIR /pagekit
VOLUME ["/pagekit/storage", "/pagekit/app/cache"]
```

前面使用 ENV 设置了 pagekit 版本号,并且设置了 WORKDIR,后面的 RUN 指令都会在该目录下进行。下面表示从 Github 上拉取相应版本的 Pagekit 压缩包。

```
RUN wget https://github.com/pagekit/pagekit/releases/download/$PAGEKIT_
VERSION/pagekit-$PAGEKIT_VERSION.zip -O /pagekit/pagekit.zip && \
    unzip /pagekit/pagekit.zip && rm /pagekit/pagekit.zip
// 完成了这些,pagekit 的 Dockerfile 基本已经写好了,然后还需要配置 Nginx 以及清理镜像:
ADD nginx.conf /etc/nginx/nginx.conf
RUN chown -R www-data: /pagekit && \
    apt-get autoremove wget unzip -y && \
    apt-get autoclean -y && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
CMD ["sh", "-c", "service php5-fpm start && nginx"]
```

使用 ADD 指令可以添加构建目录的文件到镜像中,然后清理镜像,删除“临时软件”,并清理缓存目录,最后设置 CMD 启动命令。如此 Dockerfile 就写完了。

Nginx 配置文件网址为: <https://github.com/izuolan/dockerfiles/blob/master/pagekit/nginx.conf>。

这份 Dockerfile 并不算标准,例如没有把 Nginx 与 PHP 环境分离开来,不符合 Docker 的理念。但是不管怎么说,这是一个适合入门的 Dockerfile 例子,读者可以清晰地感受到写这份 Dockerfile 时就如同在 Linux 终端下配置 PHP 环境一样,有条不紊。

6.3.3 设置自动构建

写完了 Dockerfile 后可以使用 Docker Hub 自动构建镜像,这些内容在第5章中有详细介绍,这里不再赘述。

6.4 本章小结

本章主要介绍了 Dockerfile 配置文件的编写及构建过程。在解释 Dockerfile 指令时，也提出了一些规范 Dockerfile 写法的要求，希望读者能够在“可读性”的基础上出发去写 Dockerfile。写一份高质量的 Dockerfile 并不是一件十分容易的事情。在后面的学习中还会遇到很多 Dockerfile。读者也可以建立自己的 Dockerfile 仓库，收集积累经验。

第 7 章 Docker 仓库

仓库 (Registry) 是用来集中存储 Docker 镜像的一种工具, 可以方便镜像的存储、分发、更新等管理操作。当前最大的 Docker 仓库是 Docker 官方的 Docker Hub (<https://hub.docker.com/>), 除了官方仓库, 互联网上还有数不胜数的第三方公开镜像仓库、企业私有仓库等。

本章首先介绍官方的仓库, 从最基础的使用到 Registry API、存储驱动、实现原理等, 逐步深入。此外针对国内环境, 还会介绍国内几家较为著名的第三方仓库, 还会尝试在本地搭建一个简单的私有仓库。

7.1 官方仓库 Docker Hub

Docker Hub 是来自官方的镜像仓库, 目前托管着大量镜像。值得注意的是, 在本书编写期间, Docker 公司正在逐步把 Docker Hub 迁移到 Docker Store, 这一点与前面第 2 章提到的如出一辙——Docker 不会只做一个容器服务提供商, 而是企图打造一个更完善、更庞大的“软件商店”。目前 Docker Store 提供了付费镜像的服务, 镜像加密等功能也已经进入测试阶段, 相信不久的将来, Docker 的镜像仓库会更加完善。

鉴于 Docker 公司目前没有正式推广 Docker Store, Docker Store 在很多方面也不够完善, 所以本书依旧以 Docker Hub 为例进行讲解, 相信由于兼容性问题, Docker Hub 与 Docker Store 在操作上不会有太大差别。

7.1.1 Docker Hub 登录与使用

登录 Docker Hub 有两种方式, 一种是在终端里面使用 `docker login` 登录。

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: your_user_name
Password:
Login Succeeded
```

输入密码时是看不到输出的, 实际上正在输入, 输入完毕之后回车即可。

如果没有账号, 则需要到 <https://hub.docker.com> 注册, 如图 7.1 所示。

Docker Hub 的镜像命名类似 Github 的 Git 仓库命名, 例如下面是官方 Nginx 镜像的地址, library 是官方维护的仓库之一, 其中有大量常用的软件应用。 <https://hub.docker.com/r/library/nginx/>。

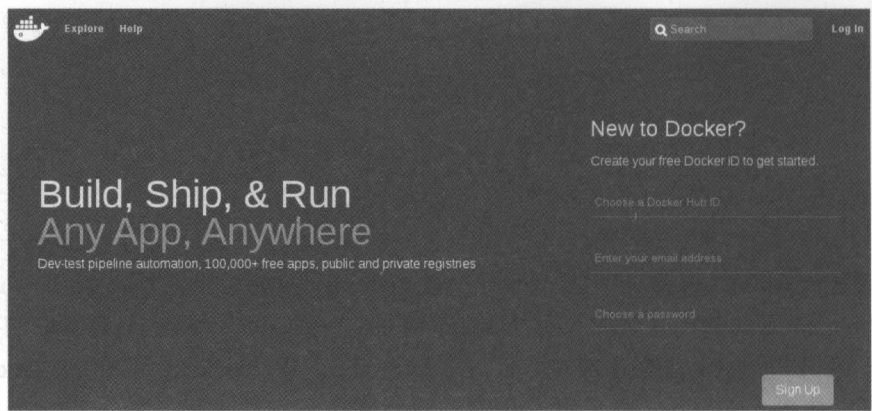


图 7.1 Docker Hub 首页

因为历史原因，可以通过下面地址访问 library 仓库，但是建议按照统一标准使用上面的地址格式。

- `https://hub.docker.com/r/_/nginx/`;
- `https://hub.docker.com/_/nginx/`。

与之对应，每个用户或者组织都有一个自己的公开页面，以 library 为例，网址为 `https://hub.docker.com/u/library`。

如图 7.2 所示，拉取镜像只需要 `https://hub.docker.com/r/library/ubuntu/` 后面的 `library/ubuntu` 部分，即：

```
$ docker pull library/ubuntu
```

而具体的标签可以在 `https://hub.docker.com/r/library/nginx/tags/` 中查看。镜像的使用说明可以阅读 Repo Info 标签页的内容。

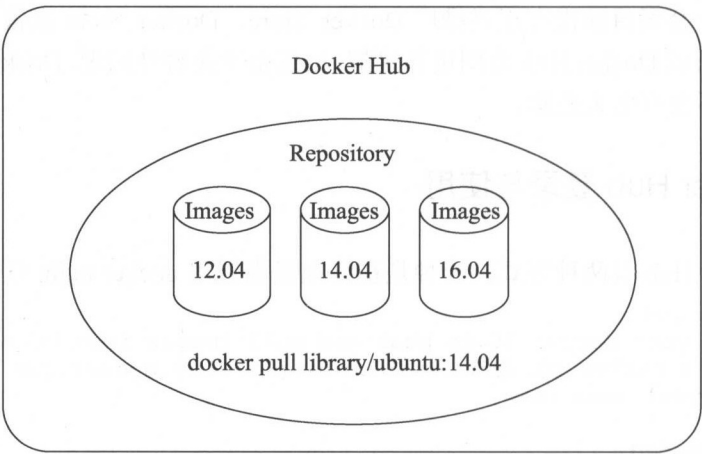


图 7.2 一张图说明仓库结构

7.1.2 Docker Hub 与 Docker Cloud

Docker 收购 Tutum 公司后，推出了 Docker Cloud，这是一个面向企业的容器云平台，

它与 Docker Hub 结合,极大方便了用户对 Docker 的使用。Docker Cloud 地址为 <https://cloud.docker.com/>。

使用 Docker Hub 的账号登录可以看到,这是一个集镜像构建、集群管理、容器编排于一体的企业服务,大部分业务是免费的,但是有一定限制。

Docker 也提供了企业版本的管理面板,但是价格不菲,国内外目前开源的功能较全的管理面板还没有,即使有也是“残缺”的社区版。目前 Docker Cloud 基本实现了 Docker 命令的可视化。

7.2 国内镜像仓库

7.1 节中介绍的搜索镜像都是从官方镜像仓库中搜寻,事实上除了 Docker Hub 上的镜像,许多第三方仓库也是有着体量不小的镜像。

同时,由于国内网络环境访问 Docker Hub 的速度不理想,使用国内镜像仓库就很有必要了,否则国内直接拉取 Docker Hub 的镜像会非常慢。下面介绍如何使用国内的镜像仓库。

7.2.1 中国科学技术大学镜像仓库

非盈利性质的镜像仓库,当然首先要介绍中国科学技术大学的 Docker 镜像仓库。要使用这个镜像仓库替换官方仓库非常简单,只需一步即可。

Docker 1.12 使用 `daemon.json` 来配置 Daemon。

```
/etc/docker/daemon.json (Linux)
%programdata%\docker\config\daemon.json (Windows)
```

在该配置文件中加入(没有该文件的话,请先建一个):

```
{
  "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"]
}
```

Docker 1.12 之前版本的配置方法,在 Docker 的启动参数中加入:

```
--registry-mirror=https://docker.mirrors.ustc.edu.cn
```

Ubuntu 用户(包括使用 `systemd` 的 Ubuntu 15.04)可以修改 `/etc/default/docker` 文件,加入如下参数:

```
DOCKER_OPTS="--registry-mirror=https://docker.mirrors.ustc.edu.cn"
```

其他 `systemd` 用户可以通过执行:

```
$ sudo systemctl edit docker.service
```

修改设置,覆盖默认的启动参数。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker -d -H fd:// --registry-mirror=https://docker.mirrors.ustc.edu.cn
```

Docker Daemon configuration file 文档地址为 <https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file>。

Docker for Windows 文档地址为 <https://docs.docker.com/docker-for-windows/#/docker-daemon>。

7.2.2 DaoCloud 镜像加速器

DaoCloud 是国内一家比较著名的 Docker 初创公司，该公司面向用户开放了一个加速仓库。镜像拉取完全免费，虽然有流量限制，但是速度非常快，地址为 <https://www.daocloud.io/mirror>。

如图 7.3 所示，安装 DaoCloud 加速器后不需要改变命令可以直接使用 `docker pull` 命令拉取。DaoCloud 需要 Docker 1.8 或更高版本才能使用，支持 Linux、Mac OS 以及 Windows 平台。

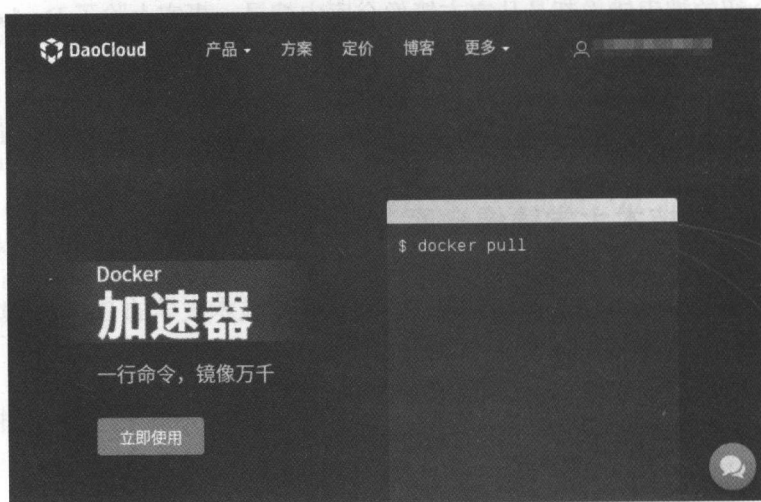


图 7.3 DaoCloud 加速器

(1) Linux 平台安装办法

Linux 平台安装脚本适用于 Ubuntu 12.04+、Debian、CentOS 6+、Fedora、Arch Linux、openSUSE Leap 42.1+，TOKEN 需要登录 Daocloud 才能获取：

```
$ curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s  
http://TOKEN.m.daocloud.io
```

(2) Mac OS 平台安装办法

右击桌面顶栏的 Docker 图标，在弹出的快捷菜单中选择 Preferences，在 Advanced 标签下的 Registry mirrors 列表中加入下面的镜像地址（TOKEN 需要登录 DaoCloud 才能获取）：

```
http://TOKEN.m.daocloud.io
```

单击 Apply & Restart 按钮使设置生效。

(3) Windows 安装办法

在桌面右下角状态栏中右击 Docker 图标，修改在 Docker Daemon 标签页中的 JSON，把下面的地址加到 registry-mirrors 的数组里（TOKEN 需要登录 DaoCloud 才能获取）：

```
http://TOKEN.m.daocloud.io
```

然后单击 Apply。

7.2.3 阿里云镜像加速器

阿里云同样也提供了 Docker 仓库，先登录阿里云，如图 7.4 所示，然后打开网址 <https://cr.console.aliyun.com/#/accelerator>。

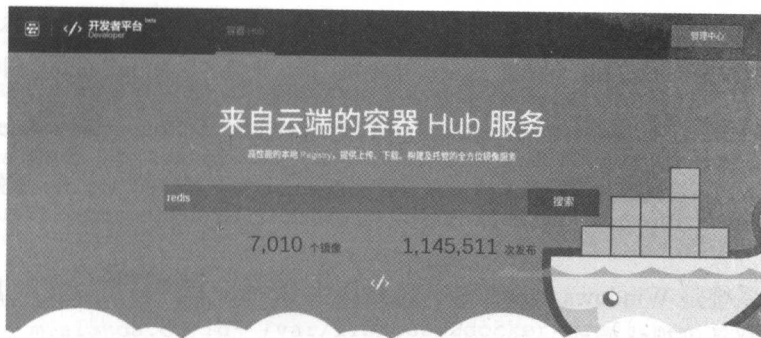


图 7.4 阿里云 Docker 仓库

一般情况下需要先开启容器服务，然后就可以在 <https://>生成一段代码.mirror.aliyuncs.com 地址中可看到下面的“专属加速地址”，如图 7.5 所示。

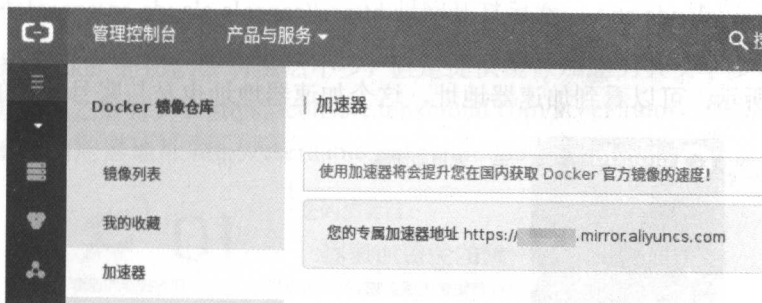


图 7.5 专属加速地址

如何配置 Docker 加速器呢？用户可以使用如下的脚本将 mirror 的配置添加到 docker daemon 的启动参数中。

如果操作系统是 Ubuntu 12.04 14.04，Docker 1.9 以上：

```
$ echo "DOCKER_OPTS=\"\$DOCKER_OPTS --registry-mirror=https://yourcode.mirror.aliyuncs.com\"" | sudo tee -a /etc/default/docker
$ sudo service docker restart
```

如果操作系统是 Ubuntu 15.04 16.04，Docker 1.9 以上：

```
$ sudo mkdir -p /etc/systemd/system/docker.service.d
$ sudo tee /etc/systemd/system/docker.service.d/mirror.conf <<-'EOF'
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// --registry-mirror=https://yourcode.mirror.aliyuncs.com
EOF
```

```
$ sudo systemctl daemon-reload sudo systemctl restart docker
```

如果操作系统是 CentOS 7 以上, Docker 1.9 以上:

```
$ sudo cp -n /lib/systemd/system/docker.service /etc/systemd/system/
docker.service
$ sudo sed -i "s|ExecStart=/usr/bin/docker daemon|ExecStart=/usr/bin/
docker daemon --registry-mirror=https://yourcode.mirror.aliyuncs.com|g"
/etc/systemd/system/docker.service
$ sudo systemctl daemon-reload sudo service docker restart
```

如果操作系统是 Windows / macOS, 并且使用 Docker Toolbox 部署 Docker, 则:

```
// 创建一台安装有 Docker 环境的 Linux 虚拟机, 指定机器名称为 default, 同时配置 Docker
加速器地址
$ docker-machine create --engine-registry-mirror=https://yourcode.mirror.
aliyuncs.com -d virtualbox default
// 查看机器的环境配置并配置到本地, 然后通过 Docker 客户端访问 Docker 服务
$ docker-machine env default eval "$(docker-machine env default)" docker
info
```

如果操作系统是 Windows / macOS 并且是原生 Docker 程序, 可以按照上面 DaoCloud 的方法设置, 都是可视化操作。

7.2.4 灵雀云镜像加速器

灵雀云也是国内一家围绕 Docker 的初创公司, 主打 CaaS (容器即服务), 先注册登录灵雀云 (<https://alauda.cn>), 然后打开网址 <https://console.alauda.cn/console/<username>/#/mirror>。

如图 7.6 所示, 可以看到加速器地址, 这个加速器地址也是与账号关联的。

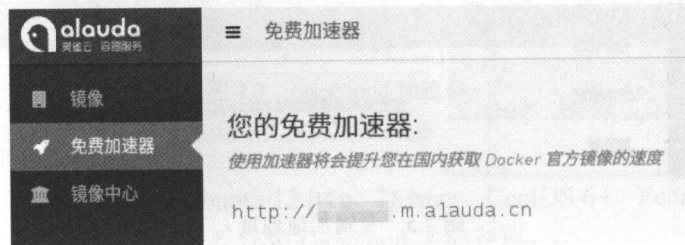


图 7.6 灵雀云加速器

配置加速器方法如下。

先将加速器加入到 Docker 配置文件中, 不同系统会有区别。

(1) Windows 系统下先启动 Boot2docker Start Shell, 然后输入:

```
$ sudo "sh -c \"echo EXTRA_ARGS='--registry-mirror=http://username.
m.alauda.cn\" >> /var/lib/boot2docker/profile\""
```

重新启动 Boot2Docker。

(2) Ubuntu 或 Debian 下直接在终端输入:

```
$ echo "DOCKER_OPTS=\"\$DOCKER_OPTS --registry-mirror=http://username.m.
alauda.cn\" | sudo tee -a /etc/default/docker
$ sudo service docker restart
```

(3) CentOS 下直接在终端输入:

```
$ sudo sed -i 's|other_args="|other_args="--registry-mirror=http://username.m.alauda.cn |g' /etc/sysconfig/docker
$ sudo sed -i "s|OPTIONS='|OPTIONS='--registry-mirror=http://stmbtfly.m.alauda.cn |g" /etc/sysconfig/docker
$ sudo sed -i 'N;s|\[Service\]\n|\[Service\]\nEnvironmentFile=-/etc/sysconfig/docker\n|g' /usr/lib/systemd/system/docker.service
$ sudo sed -i 's|fd://|fd:// $other_args |g' /usr/lib/systemd/system/docker.service
$ sudo systemctl daemon-reload
$ sudo service docker restart
```

(4) Mac OS 系统下直接在终端输入:

```
$ boot2docker ssh sudo "sh -c \"echo EXTRA_ARGS='--registry-mirror=http://username.m.alauda.cn' >> /var/lib/boot2docker/profile\""
$ boot2docker restart
```

(5) Toolbox 工具在终端输入:

```
$ docker-machine ssh default
$ sudo sed -i "s|EXTRA_ARGS='|EXTRA_ARGS='--registry-mirror=http://username.m.alauda.cn |g" /var/lib/boot2docker/profile
$ exit
$ docker-machine restart default
```

注意,上面介绍的诸多加速器只能选择一个,不能同时添加多个加速器。

7.2.5 时速云镜像加速器

国内围绕 Docker 的初创公司虽然不少,但是提供镜像加速的其实不多,时速云也是其中之一。注册登录之后打开: <https://console.tenxcloud.com/accelerator>, 如图 7.7 所示,可以看到自己的镜像加速器地址 <http://username.m.tenxcloud.net>



您的加速器:

<http://username.m.tenxcloud.net>

使用加速器将会提升您在国内获取 Docker 官方镜像的速度

图 7.7 时速云加速器

将自己的加速器加入到 Docker 配置文件中,不同系统会有区别。

(1) Ubuntu 系统:

```
$ echo "DOCKER_OPTS=\"\${DOCKER_OPTS} --registry-mirror=http://username.m.tenxcloud.net\"" | sudo tee -a /etc/default/docker
$ sudo service docker restart
```

(2) boot2docker 虚拟机:

```
$ boot2docker ssh
$ sudo su
$ echo "EXTRA_ARGS='--registry-mirror=http://username.m.tenxcloud.net'"
>> /var/lib/boot2docker/profile && exit
$ exit
$ boot2docker restart
```

(3) Toolbox 工具:

使用 docker-machine 创建 VM 的时候需要指定 registry mirror 服务器的信息。

```
$ docker-machine create \
  -d virtualbox \
  --engine-registry-mirror http://username.m.tenxcloud.net mydocker
```

细心的读者大概早就发现了，这些加速器实际上都是一个网址，因此设置加速器，其实就是在相应的地方填上地址而已。

7.2.6 网易蜂巢

与前面的镜像加速器不同，网易虽然也有基于 Docker 的云服务，但是网易蜂巢的仓库并没有对 Docker Hub 做镜像，网易蜂巢的仓库是独立的镜像仓库。可以在 <https://c.163.com/hub> 中浏览相关镜像，如图 7.8 所示。



图 7.8 网易蜂巢

使用 `docker pull hub.c.163.com/<后面部分和官方镜像地址一样>` 就可以拉取相应的镜像，但是网易蜂巢不是镜像仓库，拉取的镜像版本一般都远远滞后于 Docker Hub。

7.2.7 自建镜像加速器

前面介绍了那么多镜像加速器，读者是不是好奇这些镜像加速器是怎么搭建起来的？其实它们都是基于一个 Docker Registry 的镜像搭建的，这个项目托管在 Github 上，网址为 <https://github.com/docker/distribution>。

如果用户在海外有自己的服务器，并且经常有用不完的流量，可以把闲置的资源打造成一个镜像加速器。要实现这一功能只需要一句命令：

```
$ docker run -d -p 5000:5000 \
  -e STANDALONE=false \
  -e MIRROR_SOURCE=https://registry-1.docker.io \
  -e MIRROR_SOURCE_INDEX=https://index.docker.io \
  registry
```


这样一个简单的 Docker Hub 镜像站点就搭建好了，然后在国内的 Linux 服务器中执行一句命令就可以设置该加速地址：

```
$ docker --registry-mirror=http://服务端 ip 地址:5000 -d
```

之后在国内服务器里使用 `docker pull` 拉取镜像就会快很多了。

以上只是简单地搭建一个镜像站点，事实上 Registry 的配置还有更多可以设置的内容，例如添加用户认证，设置流量控制等，这些内容会在后面的第 18 章中将详细介绍。

7.3 私有仓库

前面介绍的 Docker Registry 都是公开的仓库，对于一些公司或者开发者来说，有些镜像是不能够公开的，如一些商业非开源软件的镜像，往往需要加密存储，为了保证安全就需要自己搭建私有仓库了。

7.3.1 搭建私有仓库

Docker 项目组公开了 Docker Registry 的源码，用户可以在 Github 上找到源代码，网址为 <https://github.com/docker/distribution>。

因为 Registry v1 已经停止开发，本书不会讨论 Registry v1 的情况，全部使用当前主流版本 Registry v2。

要部署 Registry v2，Docker 版本至少是 1.6 以上。如果用户是个人开发者，没有公开仓库分享的需求，那么搭建一个自己的仓库就非常简单了，只需要运行一个容器就可以实现私有仓库的搭建：

```
$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

7.3.2 私有仓库的使用

如没有其他问题的话，上面的私有仓库已经搭建起来了，现在可以向私有仓库推送镜像了，但是在此之前，必须先使用 `docker tag` 来给即将推送的镜像打标签，这是因为在 `docker images` 所显示的镜像，默认是从 Docker Hub 拉下来的，推送时如果不指定仓库地址，Docker 会默认推送到 Docker Hub。

例如，要把 `ubuntu:14.04` 推送到刚才搭建的私有仓库中，需要先使用 `docker tag` 改变 `ubuntu:14.04` 的镜像名称：

```
$ docker tag ubuntu:14.04 localhost:5000/ubuntu:14.04
```

改变之后就可以推送到私有仓库了，使用 `docker push` 即可：

```
$ docker push localhost:5000/ubuntu:14.04
The push refers to a repository [localhost:5000/ubuntu]
ffb6ddc7582a: Pushed
344f56a35ff9: Pushed
530d731d21e1: Pushed
24fe29584c04: Pushed
```

```
102fca64f924: Pushed
14.04: digest: sha256:4e55a1752a2030d7b43f32b1971f29862e822be9649b7dfe6b
1806245d8a8fd4 size: 1359
```

推送成功后,即使本地删除了 `localhost:5000/ubuntu:14.04` 镜像,还可以从 `localhost:5000` 这个私有仓库拉取 `ubuntu:14.04` 的镜像。

```
$ docker pull localhost:5000/ubuntu:14.04
14.04: Pulling from ubuntu
Digest:
sha256:4e55a1752a2030d7b43f32b1971f29862e822be9649b7dfe6b1806245d8a8fd4
Status: Downloaded newer image for localhost:5000/ubuntu:14.04
```

需要注意的是,上面运行的 `registry` 容器没有提供数据卷参数,所以推送的容器只存在于容器内部,如果用户删除 `registry` 容器之后所有推送到该私有仓库的镜像都会被删除,所以为了存储镜像,在实际应用中还需要在启动命令中补充数据卷参数 “`-v <宿主机本地路径>:/var/lib/registry`”,默认的镜像存储位置在 `/var/lib/registry`,用户可以自己调整参数改变存储目录,更详细的内容在后面的章节中会具体讲述如何配置一个功能完善的私有仓库,这里暂不展开介绍。

7.3.3 私有仓库安全性

前面讲述的是针对个人开发者,并且要求不高的私有仓库搭建方法。事实上,对于一家企业来说,前面介绍的私有仓库远远达不到安全、高效的要求。

比如对于企业来说,一般不会使用某个 IP 作为私有仓库的地址,而是通过反代理使用自己的域名。另一方面,前面的案例中传输过程是不加密的,所以存在中间人攻击等风险,除此之外一些基本的用户认证、负载操作和镜像存储也没有考虑,关于这部分内容会在后面章节中详细介绍,此处不再赘述。

7.4 Registry 原理

前面在搭建私有仓库时,读者可能注意到并没有账号管理的功能,这是因为在 `Registry` 镜像中并没有账号管理的功能。要完成这些功能,先要了解一个完整 `Docker Registry` 的结构。

7.4.1 Registry 组成

`Docker Registry` 有 3 个角色,分别是 `Index`、`Registry` 和 `Registry Client`。

`Index` 主要负责管理 `Docker Private Registry` 的用户信息以及认证权限,保存记录和更新用户信息(包括操作记录),以及镜像校验信息。`Index` 主要由控制单元、鉴权模块、数据库、健康检查模块和日志系统等组成,可见 `Index` 并不是一个具体存在的事物而是一个概念。

`Registry` 是镜像的仓库,然而它没有一个本地数据库,也不提供用户的身份认证,由

S3、云文件和本地文件系统提供数据库支持。此外，通过 Index Auth service（鉴权模块）的 Token 方式进行身份认证。

Docker 充当 Registry Client 来负责维护推送和拉取的任务，以及客户端的授权。

7.4.2 Registry 工作流程

了解了这 3 个角色，就大概了解了 Docker Registry 的工作流程，如图 7.9 所示为客户端发出 pull 请求下载镜像时 Registry 的工作流程，客户端向 Index 请求拉取镜像，Index 确认后返回 Token，客户端拿到 Token 向 Registry 请求拉取镜像，Registry 再向 Index 确认 Token 是否正确，无误后 Registry 允许用户拉取镜像。

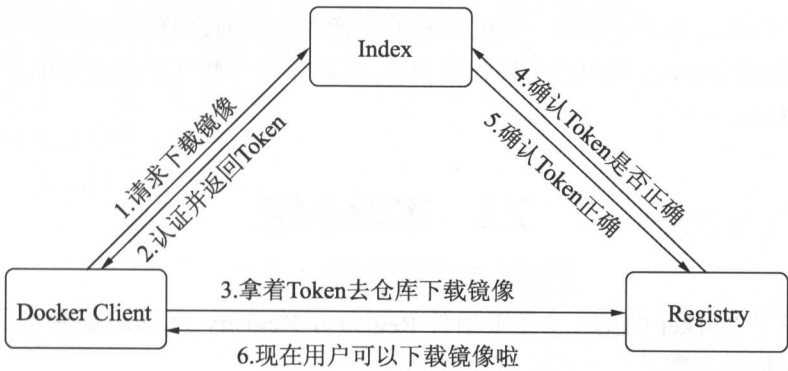


图 7.9 docker pull 工作流程

同样，当客户端提出要推送镜像到 Registry 时，需要 Index 认证，认证通过后会返回一个 Token，拿着 Token 去找 Registry，Registry 再去问 Index，Index 说是这个口令，没错，然后 Registry 才向用户开放权限允许推送，如图 7.10 所示。

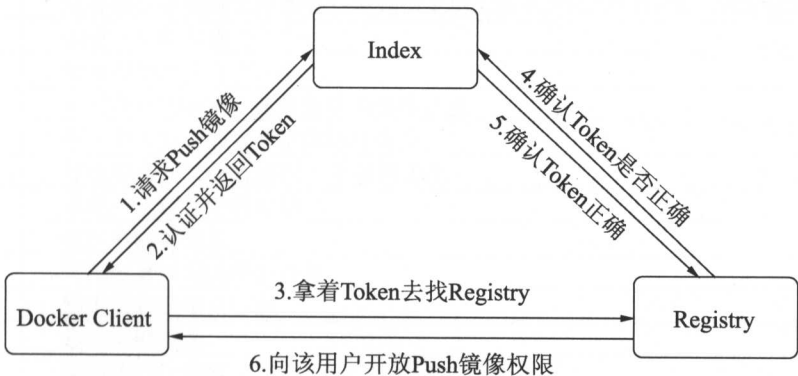


图 7.10 docker push 工作流程

稍有不同的是删除镜像的时候，Registry 收到 Client 的删除镜像的请求时，会向 Index 确认，确认无误后，Index 删除镜像元数据，并且通知 Registry 删除存储的镜像，如图 7.11 所示。

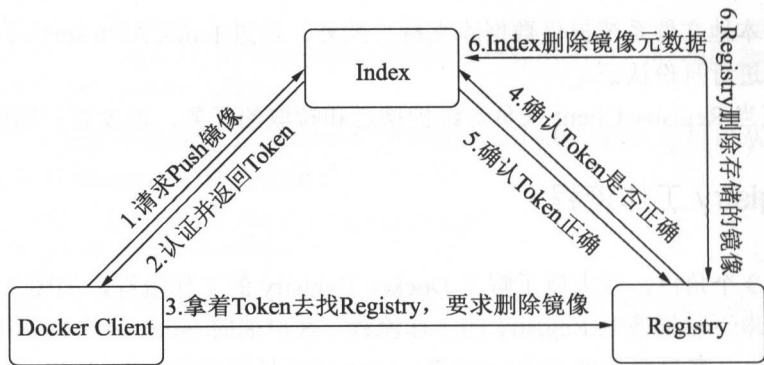


图 7.11 docker delete 工作流程

即便是一些第三方公开仓库，在匿名拉取镜像时都是通过鉴权机制发放匿名口令实现拉取的。更详细的如Index的数据库设计等内容，可以参考一些开源的仓库项目，例如Harbor等的数据库设计。

7.5 本章小结

本章介绍了 Docker 的第二个重要组件 Registry，Registry 为 Docker 镜像文件的分发和管理提供了便捷的方案。

本章还介绍了许多第三方镜像仓库，国内用户可以选择通过这些镜像仓库来存储或者拉取镜像。对于使用要求较高的个人用户还可以使用自建仓库和加速器，达到更好的体验效果。本章所介绍的私有仓库搭建过程并不完善，在后面的实战章节中会更详细地介绍如何部署一个高可用的安全性强的企业级镜像仓库。

第 8 章 Docker 容 器

容器是 Docker 的一个核心概念，容器技术的概念在第 1 章就已经讲过，Docker 容器是镜像的运行实例，它在镜像已有的文件层上添加一层可读可写的文件层，使得容器就像是一个动态的镜像。

本章围绕容器，讲解容器的操作、结构与原理。主要内容有：

- 容器的启动、停止、删除等操作。
- 容器的备份、迁移等操作。
- 容器启动过程的原理。
- 容器内部结构与实现原理。

8.1 容器基本操作

截至 Docker 1.12 版本，关于容器管理的命令如表 8.1 所示，这些命令在第 4 章中已经介绍过，在本章中会对常用的命令进行讲解，对于其他命令会在涉及时再提及。

表 8.1 容器基本命令

命 令	说 明
attach	依附到正在运行的容器
cp	从容器里面复制文件或者目录到宿主机文件系统或以STDOUT形式输出
create	创建一个新容器
diff	检查容器的文件系统变动
events	实时获得 Docker 服务器端的事件信息
exec	在一个运行中的容器里面运行命令
export	导出容器的文件系统到一个归档文件
kill	杀死一个运行中的容器
logs	获取容器的日志
pause	暂停容器内部的所有进程
port	输出容器的端口信息
ps	显示容器列表
rename	重命名一个容器
restart	重启容器
rm	删除一个或者多个容器
run	运行一个新容器
start	启动一个或多个非运行状态的容器
stats	实时显示容器的资源使用情况
stop	停止正在运行的容器
top	显示容器内正在运行的进程

命 令	说 明
unpause	恢复容器内部的所有进程
update	更新一个或者多个容器的配置
wait	阻塞直到容器停止，然后打印它的退出代码

表 8.1 中的命令大部分都不复杂，一些参数较多的命令，如 `docker run` 等可以在日后学习中慢慢积累经验。

8.1.1 创建容器

使用 `docker create` 可以创建一个容器，例如：

```
$ docker create -it ubuntu
a4ec86af07d71e1dd91b06b334ee377989abce6b51ec323c3495063835730a2c
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
a4ec86af07d7    ubuntu:latest    "/bin/bash"    20 seconds ago    Created    test
```

使用 `docker create` 创建的容器处于 `Created` 状态，这种状态类似 `Stop`，可以用 `docker start` 来启动它们。

8.1.2 启动容器

启动容器有两种情况，一种是原来没有这个容器，需要基于一个镜像启动新的容器，另一种情况是宿主机本来有一个容器，但是这个容器处于非运行状态，可以把这个非运行的容器启动起来。

启动一个新的容器可使用 `docker run` 命令，而启动一个已经存在的非运行状态的容器，使用 `docker start` 命令。

例如，下面使用 `docker run` 新建容器并启动，用这个容器来输出一句话，然后停止容器。

```
$ docker run ubuntu /bin/echo "Hello World !"
Hello World !
```

查看刚才的容器状态：

```
$ docker ps -a
```

此时上面的容器处于 `Exited` 状态，即为退出。

当使用 `docker run` 命令创建容器时，`Docker` 实际上在用户感受不到的情况下经过了一系列操作：首先检查本地是否存在这个镜像，如果没有就从镜像仓库下载；如果有就检查启动命令是否有参数冲突（例如 `-d` 和 `--rm` 不能一起用）；没有冲突时，利用本地镜像创建一个容器（类似 `docker create` 操作）；然后挂载可读写层，启动容器和一系列配置（各种资源隔离操作）；然后在应用参数值时如果遇到参数有误，启动会终止；如果没有问题则执行应用程序，执行完毕后终止容器。

以上就是一个容器启动的简单过程，关于容器详细的启动过程可以参看 8.5.2 节的内容。

下面的命令启动一个 **bash** 终端，允许用户交互操作：

```
$ docker run -it ubuntu bash
root@95df1fb2e37c:~#
```

上面的 **-it** 其实是 **-i** 与 **-t** 的缩写，很多时候这种没有参数值的参数选项都可以写到一起，例如 **-itd** 等。前面说过 **-t** 就是让 Docker 分配一个终端并绑定到容器标准输出上，而 **-i** 则是让容器的标准输出保持打开，这样就形成了一个可交互的终端界面，它是绑定到容器内部的，因此在该终端下执行的动作会在容器内部执行，就像连接虚拟机一样，例如：

```
root@95df1fb2e37c:~# ls
bin boot core dev etc home lib lib64 media mnt opt proc root
run sbin srv sys tmp usr var
```

在容器内部使用 **ps** 命令查看进程，可以看到只运行了 **bash** 程序，没有其他进程在运行。

```
root@95df1fb2e37c:~# ps
  PID TTY          TIME CMD
    1 ?            00:00:00 bash
    8 ?            00:00:00 ps
root@5233cca0b35a:/#
```

用户可以输入 **exit** 命令退出容器。这个时候容器会处于退出状态，因为前面把标准输出绑定到 **bash** 中，当 **bash** 退出时，容器自然会停止。

停止的容器可以通过 **docker start** 命令再次启动上面的容器：

```
$ docker start -i 95df1fb2e37c
root@5233cca0b35a:/#
```

启动后的容器可以使用 **docker inspect** 命令查看容器详细信息，例如查看容器 IP 地址：

```
$ docker inspect --format '{{.NetworkSettings.IPAddress}}' ${container_id}
```

命令 **docker inspect** 允许使用 Go Templates 来格式化 **inspect** 命令的输出信息。在稍后的内容中会讲到关于 Go Templates 的基本知识。

8.1.3 后台运行容器

大部分时候，运行容器都需要在后台运行较长时间，8.1.2 节那种启动容器的方式不适合大部分应用场景。

后台运行容器需要添加 **-d** 参数，例如在后台运行一个 Nginx：

```
$ docker run -d nginx:alpine
0948ef9e95091c67fec1cd9712e8e2688dcd5436c816497699aee9e333791304
```

容器启动后会返回一个唯一的容器 ID，通过 **docker ps** 命令可以看到正在运行的容器的基本信息。

使用 **docker logs** 可以查看容器的日志信息，这对于在后台运行的容器是非常重要的，有时候容器意外退出，可以通过 **docker logs** 命令来查看容器退出的原因：

```
$ docker logs <container id>
```

8.1.4 自动重启容器

前面说到容器会意外退出，说明容器在有时候会发生一些意想不到的事情，例如应用

程序内部的错误导致程序退出，从而波及容器进程，导致整个容器退出，所以需要有一个办法让容器在意外退出的时候自动重启，重新运行。

这个参数就是`--restart=always`，表示一直重启，参数默认值是不重启，`docker run` 使用这个参数启动的容器会在容器非正常退出的时候自动重启。

例如：

```
$ docker run -d --restart=always ubuntu /bin/bash
```

这个启动命令必定是运行失败的，因为没有给 `bash` 提供一个输出环境，`bash` 启动就退出导致容器跟着不断重启。使用 `docker ps` 命令可以看到容器状态一直显示：

```
Restarting (0) Less than a second ago
```

需要注意的是这里的自动重启是面向意外退出的情况，对于手动执行 `docker stop` 命令停止的容器，属于正常退出行为，并不会让容器重启。

8.1.5 停止与杀死容器

停止容器使用的是 `docker stop` 命令，有一个 `-t` 参数可以指定发送 `SIGKILL` 信号的时间。正常情况下 `docker stop` 命令向容器发送的是 `SIGTERM` 信号，该信号会使容器正常退出。

但是有时候容器会因为各种原因对 `SIGTERM` 信号没有响应，这个时候设置的 `-t` 参数就起了作用，当一定时间过后容器仍然没有停止就向容器发送 `SIGKILL` 信号，让容器强制停止。`SIGKILL` 信号类似 `kill` 命令一样会“杀”死所有正在运行的容器进程。例如：

```
$ docker stop <container id>
<container id>
```

这个时候使用 `docker ps -a` 命令可以查看到容器状态：

```
Exited (0) 2 seconds ago
```

退出代码是 0 时表示正常退出。

杀死容器的操作是 `docker kill`，这个操作可以快速停止一个容器，类似强制结束一个应用一样，这样杀死容器有可能导致数据丢失。例如：

```
$ docker kill <container id>
<container id>
```

使用 `docker ps -a` 查看容器信息可以看到退出码是 137，表示容器是非正常退出的。

```
$ docker ps -a
.....
Exited (137) 1 seconds ago
.....
```

如果是非人为的话，可以使用 `docker logs` 查看容器日志，以便定位问题。

```
$ docker logs <container id>
```

停止所有容器。

```
$ docker kill $(docker ps -a -q)
```

删除所有已经停止的容器。

```
$ docker rm $(docker ps -a -q)
```

8.1.6 删除容器

在执行上面的停止与杀死命令之后，容器不会被删除，而是以停止状态保存在宿主机中，此时容器不会占用磁盘之外的硬件资源。

如果用户需要释放磁盘资源，删除容器可以执行：

```
$ docker rm <container id>
<container id>
```

使用 `docker rm` 命令只能删除已经停止的容器，想要删除正在运行的容器，可以添加 `-f` 参数，该参数会向容器发送 `SIGKILL` 信号，例如：

```
$ docker rm -f <container id>
<container id>
```

除了 `-f` 参数，`docker rm` 命令还有 `-l` 与 `-v` 两个比较常用的参数。`-l` 参数的作用是删除容器与其他容器的关联，但会保留容器。`-v` 参数可以在删除容器的时候也删除数据卷，默认情况下容器与数据卷的生命周期是相互独立的。

```
$ docker run -v /srv:/srv -d ubuntu
$ docker rm -v -f <container id>
<container id>
```

8.1.7 查看容器信息

前面已经多次提及 `docker inspect` 命令，该命令用于获取容器/镜像的元数据，在之前的章节中也举了几个例子，还记得 `-f` 参数可以用于获取指定的数据吧，例如使用 `docker inspect -f {{.IPAddress}}` 来获取容器的 IP 地址等。

不过对于初学者来说，很容易被该特性的语法弄晕，并且少有人能将它的优势发挥出来，大部分教程都是通过 `grep` 来获取指定数据，虽然有效但比较零散混乱。本节将详细介绍 `-f` 参数，并给出一些例子来说明如何使用它。

前面说过，`-f` 的参数值其实是个 Go 模版，把读取到的容器/镜像的元数据按照模版格式返回内容。对于不熟悉 Golang 的用户来说不容易理解，其实 Golang 模版是一种模板引擎，让数据以指定的模式输出。Web 领域也有很多模版引擎，比如 `Jinja2`（用于 Python 和 Flask）、`Mustache`、`JSP` 等，看下面的示例：

```
$ docker inspect -f 'Image Id is: {{.Id}}' ubuntu:14.04
Image Id is: sha256:1e0c3dd64ccdb5d750d8d1dee705a64e40f1c49fd8859ae488853748c91cad43
```

该例子中 `{{.Id}}` 的内容其实就是截取了 `docker inspect ubuntu:14.04` 的信息：

```
$ docker inspect ubuntu:14.04
[
  {
    "Id":
    "sha256:1e0c3dd64ccdb5d750d8d1dee705a64e40f1c49fd8859ae488853748c91cad43",
    "RepoTags": [
      "ubuntu:14.04"
    ],
    "RepoDigests": [],
```

```

    "Parent": "",
    "Comment": "",
    "Created": "2016-10-13T21:13:06.201891855Z",
    "Container": "f0345d441216cc655ebbd58fd942ff6a7bcb38c73fe4320bc8b
6a5da376aea4d",
    .....

```

现在我们知道了一个简单的模板，通过修改 `Id` 这个字符串可以获取其他同级信息。例如上面的 `Id`、`RepoDigests`、`Parent` 等都是同级信息。

假如要获取二级信息呢？例如启动了一个容器如下：

```

$ docker run -d --name test abiosoft/caddy:php
$ docker inspect test
[
  {
    "Id": "900d1f3d1015836792eb8912de4ff45e7293c7d172feb32f8cdad6ab29109
c3f",
    "Created": "2016-10-23T06:07:43.358987797Z",
    "Path": "/usr/bin/caddy",
    "Args": [
      "--conf",
      "/etc/Caddyfile"
    ],
    "State": {
      "Status": "exited",
      "Running": false,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 0,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2016-10-23T06:07:46.139224733Z",
      "FinishedAt": "2016-10-23T06:07:53.037935547Z"
    },

```

现在如果想要查询容器启动时间，可以通过 `StartedAt` 获得，但是 `StartedAt` 信息属于 `State` 下面的内容，所以要这样写：

```

$ docker inspect -f '{{.State.StartedAt}}' test
2016-10-23T06:07:46.139224733Z

```

又比如获取容器退出码，确定容器状态，还可以同时显示多项信息。

```

$ docker inspect -f '{{.State.ExitCode}}' test
0
$ docker inspect -f '容器创建时间是: {{.State.StartedAt}} \
                      容器退出码是: {{.State.ExitCode}}' test
容器创建时间是: 2016-10-23T06:07:46.139224733Z 容器退出码是: 0

```

除了可以单纯使用模板外，还可以通过加入一些基本语法，例如获取所有退出码为非 0 的容器名：

```

$ docker kill test
$ docker inspect -f \
  '{{if ne 0 0 .State.ExitCode }}{{.Name}} {{.State.ExitCode}}{{ end }}' \
  $(docker ps -aq)
/test 137

```

又例如查看容器 IP 地址：

```
$ docker inspect -f '容器的 IP 是: {{.NetworkSettings.IPAddress}}' test
容器的 IP 是: 172.17.0.9
```

通过上面的几个示例可以看到，非常方便就能获得容器的数据，但是这个语法有点奇怪，没关系，下面来解释一下 Golang 模版的基本用法。只需要记住以下几点就可以了。

- `{{}}` 语法用于处理模版指令，大括号外的任何字符都将直接输出。
- “.” 表示“当前上下文”，前面提到的同级信息就表示在一个层级上，子级信息通过“.”符号可以传达下去。例如“`.NetworkSettings.IPAddress`”就表示 `NetworkSettings` 下面的 `IPAddress` 信息，`NetworkSettings` 与 `Config` 等信息属于同级。
- 允许使用函数可以重定义上下文，`with` 会重定义上下文环境。

```
$ docker inspect -f '{{.State.Pid}}' test
32289
$ docker inspect -f '{{with .State}} {{.Pid}} {{end}}' test
32289
# 使用 $ 符号可以获取 root 的上下文:
$ docker inspect -f '{{with .State}} {{$.Name}} 的 Pid 是: {{.Pid}} {{end}}'
test
```

- 使用 `index` 可以获取指定下标的数组值。

```
$ docker inspect -f '{{.HostConfig.Binds}}' test
[/home/user/srv:/srv]
# 使用 index .HostConfig.Binds 0 可以获得上面数组的第一个值
$ docker inspect -f '{{index .HostConfig.Binds 0}}' test
/home/zuolan/srv:/srv
```

- 除了 `with()`、`index()` 函数，其他很多函数也很常用。比如逻辑函数 `and()`、`or()` 可以返回布尔结果。注意，函数是不能放在中间的。

```
$ docker inspect -f '{{and true false}}' test
false
$ docker inspect -f '{{true and false}}' test
Template parsing error: template: :1:2: executing "" at <true>: can't give
argument to non-function true
```

常用的比较函数有 `eq()`(等于)、`ne()`(不等于)、`lt()`(小于)、`le()`(小于等于)、`gt()`(大于)、`ge()`(大于等于)等。

- `json()` 函数可以把信息输出为 JSON 格式。

```
$ docker inspect -f '{{json .NetworkSettings.Ports}}' test
{"2015/tcp":null,"443/tcp":null,"80/tcp":null}
```

- 最后一点，就是 `if` 语句的使用，在前面的查退出码非 0 的容器例子中就用到了 `if` 语句。注意，`{{end}}` 语句必须有，`else if` 和 `else` 按需使用。

```
$ docker inspect -f \
  '容器是{{if eq .State.ExitCode 0.0}}正常退出{{else}}非正常退出{{end}}'
test
容器是正常退出
```

更多内容可参考官方文档 <https://golang.org/pkg/text/template/>。

8.2 进入容器内部

经过前面几节内容的讲解，相信读者已经学会基本的容器操作了，现在你已经可以让一个容器启动起来了，但是容器对宿主机文件系统是完全隔离的，很多时候我们需要查看容器的文件系统，而使用 `-d` 参数之后，我们只能通过 `docker ps` 看到容器的运行状态。怎么进入到一个正在运行的容器里面呢？

8.2.1 使用 attach 进入容器

使用 `docker attach` 属于 Docker 的自带命令，该命令依附到正在运行的容器。

```
$ docker run -dit --name test ubuntu:14.04
1cc65411a3cb8e4d99c6f632eaf60168162ddf4a838ed15fc49717c1a5f51662
$ docker attach test
^C
root@1cc65411a3cb:/#
```

在前面的章节中介绍过 `docker attach` 的用法，其中提到不要使用 `exit` 命令（或者 `Ctrl + C`），那样会使 Docker 容器停止。要退出容器，可使用 `Ctrl + P`，然后再使用 `Ctrl + Q`，即可退出容器的虚拟终端，此时容器进程还在运行。

使用 `attach` 命令有时候并不方便。当多个窗口同时 `attach` 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时，其他窗口也无法执行操作了。

下面以 `top` 命令为例，在多个终端下查看输出：

```
$ docker run -d --name topdemo ubuntu:14.04 /usr/bin/top -b
bb83ff89a1003727e1d381821cb4cd5442e6c031c56006178c036f0773dd1a2d
$ docker attach topdemo
top - 06:24:03 up 20:50, 0 users, load average: 0.29, 0.26, 0.35
Tasks: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu(s): 11.6 us, 5.2 sy, 0.0 ni, 82.5 id, 0.8 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12193136 total, 7539804 used, 4653332 free, 144836 buffers
KiB Swap: 1999868 total, 21400 used, 1978468 free. 3836824 cached Mem

  PID USER  PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TIME+  COMMAND
    1  root   20   0 19748 2256 1992  R   0.0   0.0   0:00.01 top
.....
top - 06:24:21 up 20:50, 0 users, load average: 0.20, 0.24, 0.35
Tasks: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.0 us, 1.0 sy, 0.0 ni, 96.6 id, 0.3 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 12193136 total, 7539688 used, 4653448 free, 144864 buffers
KiB Swap: 1999868 total, 21400 used, 1978468 free. 3836232 cached Mem

  PID USER  PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TIME+  COMMAND
    1  root   20   0 19748 2256 1992  R   0.0   0.0   0:00.01 top^C
$
```

此时打开其他终端，输入 `docker attach topdemo` 看到的信息是一样的。相反，从上面看出，如果要实时查看容器日志信息等行为，使用 `docker attach` 是最方便的。

8.2.2 使用 exec 进入容器

`docker exec` 命令用于进入容器内部进行操作，与 `attach` 原理不一样，`docker exec` 命令可以像使用 SSH 登录服务器一样操作容器。

`docker exec` 命令的参数有以下几个。

- `-d` 分离模式:在后台运行的命令。
- `-i` 交互模式。
- `-t` 分配一个 TTY。
- `-u` 指定用户和用户组，格式: `<name|uid>[:<group|gid>]`。
- `--privileged` 参数会分配一个特权给 `tty` 界面，相当于拥有了宿主机的 `root` 权限，慎用。

下面以前面的 Ubuntu 容器为例，先启动容器，然后使用 `exec` 命令进入容器：

```
$ whoami
ubuntu
$ docker exec -it ubuntu bash
root@2ec3c5a455e2:/# whoami
root
root@2ec3c5a455e2:/#
```

可以看到使用 `exec` 命令进入容器内部就如同进入另一台机器一样，可以灵活操作，并且使用 `exit` 命令退出时，不会像 `attach` 命令那样导致容器停止，所以非常适合在容器内部操作。此外，每个 `docker exec` 命令都会分配一个不同的 `tty` 给用户，所以不会像 `docker attach` 命令那样有阻塞。

8.2.3 使用 nsenter 进入容器

`nsenter` 是一个第三方工具，`nsenter` 的功能远不止进入容器操作那么简单，它的名字已经解释了其原理 Namespace Enter，安装 `nsenter` 非常简单：

```
$ docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
```

这个容器会把二进制的 `nsenter` 复制到 `/usr/local/bin` 目录下，使用 `nsenter` 需要先获取容器的 PID：

```
$ PID=$(docker inspect --format {{.State.Pid}} <container_name_or_ID>)
```

然后就可以使用 `nsenter` 进入容器了：

```
$ nsenter --target $PID --mount --uts --ipc --net --pid
```

每次都要查看容器的 PID 才能进入容器内部，比较麻烦，因此可以用 `shell` 脚本来完成这一步，`nsenter` 的作者写了一个脚本在 Github 上面，地址为 <https://github.com/jpetazzo/nsenter/blob/master/docker-enter>。

用户复制内容到本地保存为 `docker-enter`，然后使用 `chmod +x docker-enter` 赋予文件执行权限，如此就完成了自动化进入容器的脚本，使用：

```
./docker-enter <container_name_or_ID>
```

即可进入容器。`nsenter` 不仅可以进入容器，还可以向正在运行的容器添加数据卷，这是原生 Docker 命令不支持的，更多信息可以查看 <https://github.com/jpetazzo/nsenter>。

8.3 导出和导入容器

就像镜像可以导出、导入一样，容器也可以导出、导入，`docker save` 和 `docker load` 命令是镜像导出、导入的一对命令，而 `docker export` 和 `docker import` 命令则是容器导出、导入的一对命令。

8.3.1 导出容器

导出容器是指把容器导出到一个归档文件中，不管容器处于运行还是停止的状态都可以导出容器。导出容器会把容器的可读可写的文件层也打包进去，但是不会把 Volume 的内容囊括进来。例如：

```
$ docker run -v ~/srv:/srv -d --name test abiosoft/caddy:php
1ee87a4a0be13a143ae1d13bfe5d534dda52b5381e46ec39b4249afb5244ae5d
```

在容器内部创建一个文件 `test`：

```
$ docker exec -it test sh
/srv # touch test
/srv # exit
```

导出容器：

```
$ docker export test > test.tar
$ ls
test.tar
```

8.3.2 导入容器

导入容器并非是把容器导入为容器，导入的容器会变成一个镜像，启动这个镜像才可以恢复容器，这一点有点像 `docker commit` 操作。

一般情况下直接使用导入 `tar` 包即可：

```
$ docker import test.tar
$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED      SIZE
<none>          <none>      df05b3fd976e  2 seconds ago  84.89 MB
```

这样导入容器所生成的镜像没有名称与标签，需要手动打标签。

如果希望在导入时打标签，可以使用管道导入：

```
$ cat test.tar | docker import - username/test:latest
REPOSITORY      TAG          IMAGE ID      CREATED      SIZE
username/test    latest      c41e33ea48d8  3 seconds ago  84.89 MB
```


还可以使用网络地址直接导入：

```
$ docker import https://example.com/container.tar
```

这样就省去了手动下载传输的麻烦。

甚至可以从目录导入：

```
$ sudo tar -c . | docker import - /path/Containerdir
```

 **注意：**在这个例子中特别使用了 `sudo`，是因为在非 `root` 环境下直接导入一个目录有可能不能保留原有文件的权限属性，所以使用这个命令导入需添加权限。

此外，在导入过程中使用 `--change` 还可以改变 `Dockerfile` 中的指令，使用 `--message` 可以添加 `commit` 信息。

```
$ cat test.tar | docker import --message "导入容器并打标签" - username/test:latest
```

使用 `--change` 参数可以在原有的 `Dockerfile` 后面追加指令，例如：

```
$ cat test.tar | docker import --change "CMD cat /etc/hosts" - username/test:latest
sha256:1b146a093665c0d026175c7f4cab99b4479d961abee4cb10d8382d169d695758
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
username/test        latest       1b146a093665     3 seconds ago   84.89 MB
$ docker run --rm username/test
127.0.0.1            localhost
::1                  localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.10         683af9e2bd51
$
```

原本容器是一个 `PHP` 容器，经过 `--change` 修改之后执行的是 `cat /etc/hosts` 命令。

因为 `docker load` 与 `docker import` 有些类似，这里提一点要注意的地方，既然两者都可以导入文件到本地镜像库，那么区别在哪里？

很明显，使用 `docker export` 导出的容器属于快照一样，它会丢失原来镜像的历史记录与元数据，而 `docker save` 保存的镜像保留了全部信息。

8.4 容器结构

Docker 容器是 Docker 镜像的运行态的体现。概括而言，就是在 Docker 镜像之上，运行进程。既然这样，容器的内部结构必定与镜像结构十分类似，本节就对容器结构进行介绍。

8.4.1 容器格式是什么

随着容器技术发展，Linux 基金会于 2015 年 6 月成立 OCI（Open Container Initiative）组织，该组织旨在围绕容器格式和运行时制定一个开放的工业化标准。该组织一经成立便得到了包括谷歌、微软、亚马逊、华为等一系列云计算企业的支持。

而前面提到的 `runC` 就是按照该开放容器格式标准（Open Container Format, OCF）制

定的一种具体实现，runC 基于 Docker 公司贡献出来的 Libcontainer 项目发展而来。制定容器格式标准使得容器不受上层结构的绑定，如特定的客户端、编排栈等，同时也不受特定的云服务商或项目的绑定，也就是说不限于某种特定操作系统、硬件、CPU 架构、公有云等。

该标准的规范文档在 GitHub 上托管，地址为 <https://github.com/opencontainers/specs>。

标准化容器有两大特点，分别是操作标准化与工业自动化。

所谓操作标准化，指的是创建、启动、停止容器使用一套标准，只要实现了接口都可以操作容器，还包括使用标准文件系统工具复制和创建容器快照，使用标准化网络工具进行下载和上传。

而工业自动化，主要有两方面，一方面是交付流程，标准容器技术的软件分发可以达到工业级交付标准，另一方面是容器自动化，与操作内容无关，与平台无关，实现标准接口就可以实现容器操作自动化。

要实现这两方面，就需要做到内容无关、基础设施无关，也就是指不管针对具体的容器内容是什么，容器标准操作执行后都能产生同样的效果。如容器可以用同样的方式上传、启动，不管是 PHP 应用还是 MySQL 数据库服务；而基础设施无关是指任何设备都应该支持容器的各项操作。

其实，开放容器格式（OCF）标准的实现要求非常宽松，它并不限定具体的实现技术也不限定相应框架，目前已经有基于 OCF 的具体实现，相信不久后会有越来越多的项目出现。

下面是一些比较著名的项目，以供参考：

- 容器运行时 opencontainers/runc，即前面所讲的 runC 项目，是后来者的参照标准。
- 虚拟机运行时 hyperhq/runv，基于 Hypervisor 技术的开放容器规范实现。
- 测试 huawei-openlab/oct 基于开放容器规范的测试框架。

8.4.2 容器内部结构

Docker 容器的文件系统可以说大部分由 Docker 镜像来提供。前面说过，容器是在镜像的文件层之上新建一层可读写层，因此容器内部大部分是镜像的内容。如图 8.1 所示为容器的内部结构。

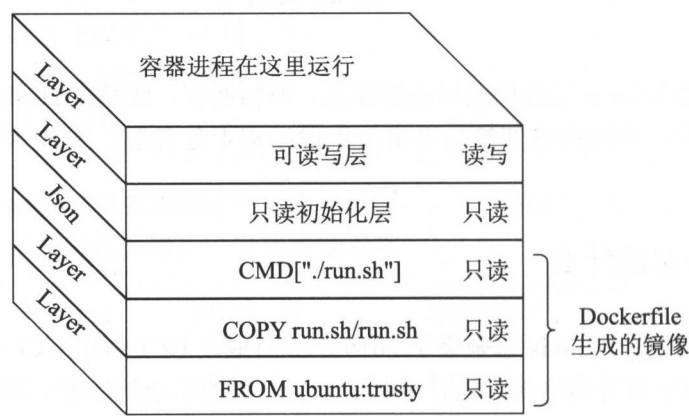


图 8.1 容器结构

图 8.1 展示了 Dockerfile、Docker 镜像与 Docker 容器三者的关系。在图 8.1 中我们假设了一个 Dockerfile 如下的镜像：

```
FROM ubuntu:trusty
COPY run.sh /run.sh
CMD ["./run.sh"]
```

以上 Dockerfile 中的每一条命令，都在 Docker 镜像中以一个独立镜像层的形式存在，这也提示了我们在构建镜像时通过减少镜像层是一个不错的控制体积的办法。

图 8.1 中 Dockerfile 组成的几层文件层实际上就是一个镜像的文件层，镜像每一层对应一条 Dockerfile 指令。

然后就是 Docker 容器，Docker 容器是 Docker 镜像的运行态的体现。前面曾提及，Docker 容器的文件系统中不仅包含 Docker 镜像，图 8.1 中的顶上两层，就是 Docker 为 Docker 容器新建的内容，而这两层不属于 Docker 镜像的结构。

这两层分别为 Docker 容器的初始层（Init Layer）与可读写层（Read-Write Layer），初始层中大多是初始化容器环境时，与容器相关的环境信息，如容器主机名、主机 host 信息以及域名服务文件等。

而可读写层，是 Docker 容器内的进程拥有读写权限的文件层，其他层对进程而言都是只读的（Read-Only）。AUFS 文件系统下，可读写层用到了写时复制（Copy-on-Write）的技术。需要注意的是，数据卷的文件也会挂载到可读写层，虽然 Docker 容器在可读写层可以看到数据卷的内容，但那仅仅是挂载点，真实内容位于宿主机上。

8.5 本章小结

容器是 Docker 的一个重要组成部分，它是 Docker 快速、高效的服务性能的基础。本章通过学习 Docker 容器的基本操作，掌握了基本的容器操作方法，然后又介绍了容器内部的结构，相信读者对容器的整个生命周期有了基本认识，在以后的章节中会在此基础上进一步展开介绍，如容器高可用等应用场景的实例介绍。

第9章 数 据 卷

前面已经把 Docker 的大部分基本知识讲完了，本章是第 8 章的扩展，着重介绍容器的数据卷管理功能。数据卷在 Docker 容器技术中扮演着非常重要的角色，可以说只要产生数据的 Docker 应用都需要用到数据卷。

本章的学习内容主要有：

- 理解数据卷原理。
- 学会挂载数据卷。
- 使用数据卷容器。
- 卷插件的使用。

9.1 数据卷是什么

从前面的学习中，我们知道了 Docker 启动之后，容器内的文件和宿主机是隔离开的，如果不使用 `docker commit` 操作提交容器为镜像把数据保存下来的话，数据就会因为容器的删除而丢失。而在前面学习构建镜像的过程中，已经明确说明了尽量不要使用 `docker commit` 提交镜像，因为会导致镜像无法通过 Dockerfile 复现，不利于迁移、重新构建等情况。

为了可以保存数据，又不至于破坏镜像的可复现特性，Docker 提出了数据卷的概念。Docker 的卷概念有两种，即数据卷和数据卷容器。

9.1.1 数据卷介绍

数据卷简单来讲就是一个目录，它是由 Docker daemon 挂载到容器中的，因此数据卷并不属于联合文件系统。也就是说数据卷里面的内容不会因为容器的删除而丢失。举一个比较形象的例子就是数据卷就像一个 U 盘，可以连接计算机，即使计算机硬盘坏了，只要不影响 U 盘，U 盘里面的数据并不会随着硬盘损坏而丢失。

这一特性也说明，如果挂载了数据卷，使用 `docker commit` 提交时并不会把数据卷里面的内容提交到镜像中。

虽然一开始数据卷概念的提出只是出于数据持久化的考虑，但很快人们发现数据卷的功能远不止于此，比如可以让两个容器使用同一个数据卷，也就是数据卷共享。

9.1.2 数据卷容器介绍

前面提到如果要想两个容器共享数据卷，最快的办法就是让它们挂载同一个目录作为数据卷即可。但是有时候不能把数据保存在宿主机中，或者数据要备份、迁移等，保存在宿主机很不方便，于是就有了专门用来存放数据的数据卷容器。

数据卷容器并不需要运行任何应用，只是一个存放数据的容器，允许其他容器挂载就可以了。数据卷容器不是特指某一个具体的镜像启动的容器，事实上每一个容器都可以担当数据卷容器的功能。

不要为了数据容器而使用“最小的镜像”，如 Busybox 或 Alpine，只使用数据库镜像本身就可以了。因为已经拥有该镜像，所以并不需要占用额外的空间。

另外一个要注意的就是不要运行数据容器，这纯粹是在浪费资源，只要数据卷容器存在，数据卷就不会消失，不管容器处于什么状态。

目前 Docker 数据卷还存在一些问题，比如对数据卷的生命周期的管理，对远程数据卷的挂载等都有不足，但是从最新实验版的 Docker 来看，第三方插件的子命令 `docker plugin` 已经趋向成熟，相信在不久的将来会有第三方插件补充这些功能（目前也有一些数据卷管理工具，但都是通过第三方命令实现的）。

9.2 为容器挂载数据卷

了解了数据卷的作用，本节将通过几个例子来说明数据卷的挂载操作。

9.2.1 挂载数据卷

在使用 `docker run` 或者 `docker create` 命令时，可以指定 `-v` 参数来添加数据卷，这个参数可以使用多次，这样就可以挂载多个数据卷了，同时数据卷还可以使用 Linux 的软链作为数据卷（受制于容器程序与真实目录的文件系统）。

这里以一个 Alpine 镜像为例：

```
$ docker run -d -v /vol --name=volume alpine
```

现在已经为 `volume` 容器挂载了一个数据卷，位置是 `/vol`。

但是这样启动挂载的数据卷不容易管理，可以使用 `docker inspect` 来查看数据卷位置：

```
"Mounts": [
  {
    "Name":
      "998918e8a60aff1c619ba7b60781ed38f530193cc01d6166a4c706bf6abf434a",
    "Source":
      "/var/lib/docker/volumes/998918e8a60aff1c619ba7b60781ed38f530193cc01d61
      66a4c706bf6abf434a/_data",
    "Destination": "/srv",
    "Driver": "local",
    "Mode": "",
    "RW": true,
```

```

    "Propagation": ""
  },
  ],

```

如果删除容器:

```
$ docker rm -f test
```


然后去看该数据卷目录会发现,数据卷还在!这本来应该算是好事,避免了丢失数据,但是这样也让用户很难管理数据卷。

一个比较好的方法是向宿主机映射目录,映射目录很简单:

```
$ docker run -d -v /vol:/vol --name=volume alpine
```

这样就不用担心容器删除后数据卷还在磁盘中占用空间了。目前数据卷保存在了宿主的/vol目录。

挂载数据卷到本地不仅仅可以保存数据卷内容,在很多时候还有“奇效”,比如把程序源代码目录挂载到容器,在容器里面编译,编译成功直接在宿主机下就可以找到编译好的项目,保证宿主机的清洁与稳定。

 **注意:** 数据卷的路径必须是绝对路径。另外,删除容器时可以通过指定 `docker rm -v <container ID>` 参数删除数据卷,但是一般情况下建议映射目录到指定目录下更容易管理。

9.2.2 挂载数据卷容器

上面的挂载本地目录到容器虽然解决了数据持久化的问题,但是在迁移上还是很麻烦,比如多个容器之间共享数据卷需要迁移时,使用挂载宿主机的文件夹的方法迁移起来就会显得很麻烦,所以为了管理数据卷,可以启动一个容器专门用来存放数据。

```
$ docker run -d -v /var/lib/mysql --name=mysql_volume mysql /bin/true
```

上面就启动了一个数据卷容器, `/bin/true` 是为了覆盖原有进程并防止容器退出。

然后启动其他容器并挂载数据卷容器。

```
$ docker run -d --volume-from mysql_volume --name db1 mysql
$ docker run -d --volume-from mysql_volume --name db2 mysql
```

甚至还可以通过 `db1` 来挂载到后续启动的容器中:

```
$ docker run -d --volume-from db1 --name db3 mysql
```

这样即使删除上面4个容器中的3个,数据卷也不会丢失。即使4个容器都删除了,数据卷也不会消失,因为数据卷还会保存在 `/var/lib/docker/volumes/` 的某个目录下,这需要事先通过 `docker inspect` 命令查看 volume ID。

Docker 目前可以使用 `docker volume` 子命令管理数据卷:


```
$ docker volume ls
DRIVER          VOLUME NAME
local           0646dbfb35a10608ff42d5dc9b31956a3c2be42ea619e0506950
fd37c0968007
local           0e62ad0fbbdc1737caeede96c035992a0e148d600045ed600fce
3481873f1008
local           196fb397ef698763b00df3861993194a8efd11b0f93af82db531
```


```

b9d2da9d1b2f
local      1a8b77d173bcf8685a0b6c9ab658b4632da3acd55fa97832b21511
07c1074942
local      2e7451e3fb210a3b3914830d81551b20cd3f0eccb6b49e20630b29
5bdf05a60c
local      373bd2ba8268ab17724a70806a234369b152210cc224154c4451c9
d239b09830
local      385f7d52cc285a288b3f568380fe3473c746b29ec3ad0433993489
bf28e67315
local      38d72db0c7eb257aa61f164449a45051d6b8aec741114a527a0069
894141f46c
local      3c24c17cfaa8d5133fe01fdff5459f3223f54a79218ec859b38861
cec0643b4e
.....

```

但是 docker volume 体验很不友好，所以建议在指定数据卷时注意指定挂载目录，避免日后找不到数据卷删除。

 **提示 1:** 使用 -v 参数时还有一个小细节可以定制，例如 -v /vol:/vol:ro 表示容器对 /vol 目录只读不可写，而 -v /vol:/vol:rw 表示可读可写，这个小改动不仅适用于目录也适用于单个文件。在一些不希望改变容器配置的场景中可以用到，这个办法可以有效保护宿主机的文件系统。

 **提示 2:** 如果删除含有数据卷的容器，在删除容器时没有使用 -v 标志，这些数据卷会成为 dangling 状态。

显示所有没有挂载到容器上的数据卷：

```
$ docker volume ls -f dangling=true
```

删除这些 dangling 状态的数据卷：

```
$ docker volume rm <volume name>
```

9.2.3 数据卷挂载小结

为了更直观地了解数据卷挂载的操作，下面以一系列操作来做个实验，逐一验证数据卷挂载的各种情况。

首先是当本地不存在该文件，而容器内存在该文件的情况，尝试把不存在的文件挂载到存在该文件的容器中。以一个 Alpine 镜像为例，这里把一个修改后的 Alpine 镜像打了新标签，叫做 volume_test：

```

// 本地目录不存在 test 文件
$ docker run --name=test -v ~/test.txt:/etc/hosts -d volume_test
0cba2e50229df7508c616bd456c4ab131f2fela88385c34f8a5876fbc577b176
docker: Error response from daemon: oci runtime error: rootfs_linux.go:53:
mounting "/var/lib/docker/devicemapper/mnt/6b83c07ebedcb828f34cac69eac5a
85ce3a5f59ele8688c8dae40198671d0ecb/rootfs/etc/hosts" to rootfs "/var/
lib/docker/devicemapper/mnt/6b83c07ebedcb828f34cac69eac5a85ce3a5f59ele8
688c8dae40198671d0ecb/rootfs" caused "not a directory".
// 启动容器失败

```

然后是把本地不存在的文件夹挂载到容器内存在的文件夹中，在 volume_test 镜像中存在一个 /srv 的文件夹，文件夹里面有一个 index.php 文件。

```
// 本地目录不存在 srv 文件夹
$ docker run --name=test -v ~/srv:/srv -d volume_test
c71cf1cfa4932e3e398a7d6c4e2ae94f915b832f5506e374aedb19af4cblac62
// 启动正常，但是进入容器后发现目录被清空
$ docker exec -it test sh
/srv # ls
/srv #
```

上面两个命令已经告诉我们，数据卷的挂载是通过把本地的目录覆盖到容器中的。也就是说，当宿主机文件不存在时，不能挂载；当文件夹不存在时，挂载到容器会用一个空文件夹覆盖容器原有目录。

继续假设宿主机存在文件，容器内不存在该文件：

```
// 本地目录存在 test.txt 文件
$ docker run --name=test -v ~/test.txt:/srv/test.txt -d volume_test
2d6853c10643a735ae3d7f3aaac8c6344f9c75170e531f613d08db7cdf484e54
// 容器内存在 /srv 文件夹，里面原本有一个 index.php
$ docker exec -it test sh
/srv # ls
index.php test.txt
/srv #
// 可以看到文件挂载成功
```

接下来是宿主机存在文件夹，容器不存在该文件夹，宿主机的 `test` 文件夹里面存在一个 `hello` 文件。

```
$ docker run --name=test -v ~/test:/srv/test -d volume_test
c935ffa0d9fc5e5ac8f213a51a878e71056472b0597d2e385a29e5c748012958
// 进入容器，查看是否存在 test 文件夹，以及文件夹里面是否有 hello 文件
$ docker exec -it test sh
/srv # ls
index.php test
/srv # cd test/
/srv/test # ls
hello
/srv/test #
```

前面两个例子说明容器内部如果不存在文件，宿主机直接挂载。

接下来假设宿主机存在 `test` 文件夹，而容器内部存在的是名为 `test` 的文件，这样挂载会怎样呢？

```
$ docker run --name=test -v ~/test:/srv/test -d volume_test
385bc78e5333460dallf04535da27a3fd226df218f95c970ff2dd5609b17f816
docker: Error response from daemon: oci runtime error: rootfs_linux.go:53:
mounting "/var/lib/docker/devicemapper/mnt/fd5c42e844c3550d1a372ed939ed5
7f90dcacbd375dfed1bedfbb71ef6f3f185/rootfs/etc/hosts" to rootfs "/var/
lib/docker/devicemapper/mnt/fd5c42e844c3550d1a372ed939ed57f90dcacbd375d
fed1bedfbb71ef6f3f185/rootfs" caused "not a directory".
```

上面的情况不出意外是启动错误，然后假设宿主机是文件夹，容器也是文件夹，两个文件夹里面的内容不一样，宿主机内部有一个 `hello` 文件，容器的文件夹里面有一个 `index.php`。

```
$ docker run --name=test -v ~/srv:/srv -d volume_test
3aec30122bd7010c694e0ff8b77f7b7b6bb6f850c258786db125313060fad43b
$ docker exec -it test sh
/srv # ls
hello
/srv #
// 可以看到，宿主机文件夹会覆盖容器内部的文件夹
```

假设宿主机有一个 test.txt 文件，里面写着 Hello World，而容器里面也存在一个 test.txt 文件，里面写着 Hi World，现在挂载文件。

```
$ docker run --name=test -v ~/test.txt:/srv/test.txt -d volume_test
047cbfe45b5bc868c864fe94f7a22643d52b644947f40260097dbb579de56c5c
$ docker exec -it test sh
/srv # cat /test
Hello World
/srv #
// 宿主机会覆盖容器的文件
```

最后一种情况，宿主机存在文件 test.txt，而容器内部存在一个 test 的文件夹，现在把文件挂载到文件夹中。

```
$ docker run --name=test -v ~/test.txt:/test -d volume_test
59b5fd74a1e9e17aa2a6a9be7900b16c7dd4b3c424a4fa72a7671fa1c51bdf69
docker: Error response from daemon: oci runtime error: rootfs_linux.go:53:
mounting
"/var/lib/docker/devicemapper/mnt/b201054ed36a189b5abb599082d0b5bcbe31d
07611a0985deefd79d1221447fd/rootfs/home" to rootfs "/var/lib/docker/
devicemapper/mnt/b201054ed36a189b5abb599082d0b5bcbe31d07611a0985deefd79
d1221447fd/rootfs" caused "not a directory".
// 启动失败
```

如表 9.1 总结了以上例子中的规律。

表 9.1 数据卷使用总结

宿主机文件	容器内文件	启动参数（加粗表示不存在）	容器启动情况
不存在	文件	-v ~/test.txt:/etc/hosts	启动错误
不存在	文件夹	-v ~/srv:/srv	启动正常
文件	不存在	-v ~/test.txt:/srv/test.txt	启动正常
文件夹	不存在	~/test:/srv/test	启动正常
文件夹	文件	~/test:/srv/test	启动错误
文件夹	文件夹	-v ~/srv:/srv	启动正常
文件	文件	-v ~/test.txt:/srv/test.txt	启动正常
文件	文件夹	-v ~/test.txt:/test	启动错误

9.3 备份、恢复、迁移数据卷

使用数据卷管理数据时，可以很方便地对数据进行备份和迁移。

9.3.1 备份数据卷

最简单的备份情况是先创建一个临时容器，并挂载要备份的数据卷容器（目录是 /db_data），再挂载数据卷/host_backup 目录到容器/container_bakcup 目录，在容器中执行备份，比如压缩数据卷容器的文件，然后放到/container_data 目录，这样/host_backup 也存在该文件了，也就完成备份了。

```
$ docker run \
  --volumes-from db_data \
  -v /host_backup:/container_backup \
  ubuntu tar cvf /container_backup/backup.tar /db_data
```

9.3.2 迁移、恢复数据卷

先创建一个数据卷容器，数据卷目录名要与备份的一样：

```
$ docker run -v /db_data --name db_data2 ubuntu
```

再将备份文件恢复到数据卷容器中：

```
$ docker run \
  --volumes-from db_data2 \
  -v /host_backup:/container_backup \
  ubuntu tar xvf /container_backup/backup.tar
```

解压目录移动到/db_data 即可完成迁移。或者使用 `docker inspect` 查看 volume ID，然后移动该目录到其他地方。

9.4 容器数据卷扩展

Docker 的 volume 本质上是一个目录，而这个目录实际上是一个挂载点，在容器创建时这个挂载点会被挂载一个宿主机的目录，这个目录可以是以 volume ID 为名的文件夹，也可以是启动参数中指定的目录。挂载之后容器与宿主机显示的目录会表现为同一个目录，独立于容器内的 rootfs。

9.4.1 卷插件介绍

卷插件机制是在 Docker 1.8 的时候引进来的，Docker 传统的卷管理只能挂载本机目录到容器中，数据的备份、同步、迁移都是个挑战，因此需要第三方的插件来管理容器中的数据。

目前 Docker 社区有不少的卷插件，比较著名的有 Flocker、Convoy、GlusterFS、Keywhiz、REX-Ray 等，它们各自都有自己的特点。

本章中只介绍 Convoy 与 Flocker 插件，其他插件读者有兴趣可以去了解。

9.4.2 Convoy 的使用

Convoy 是一个单节点卷管理插件，提供创建、删除、备份、还原数据卷等功能。由于 Convoy 是单节点插件，因此对卷的迁移和共享的支持不是很好。下面来看如何安装 Convoy 插件。

安装 Convoy 的方法非常简单。

```
$ wget https://github.com/rancher/convoy/releases/download/v0.5.0/convoy.tar.gz
```



```
$ tar xvf convoy.tar.gz
$ sudo cp convoy/convoy convoy-pdata_tools /usr/local/bin/
$ sudo mkdir -p /etc/docker/plugins/
$ sudo bash -c 'echo "unix:///var/run/convoy/convoy.sock" > /etc/docker/
plugins/convoy.spec'
```

这样 Convoy 就安装好了，然后启动 Convoy Daemon（下面是示例）。

```
$ sudo convoy daemon \
  --drivers devicemapper \
  --driver-opts dm.datadev=/dev/loop5 \
  --driver-opts dm.metadatadev=/dev/loop6
```

Convoy 支持多种存储驱动，包括 Device Mapper、VFS、EBS 等（可以把 NFS 挂载到 VFS 目录下，实现跨主机存储和共享），这里以 VFS 为例启动 Convoy。

```
$ sudo convoy daemon --drivers vfs --driver-opts vfs.path=/vol
```

启动完成后就可以使用 Convoy 了，在 docker run 中加入以下几个参数即可。

```
$ docker run -it -v vol_test:/vol_test --volume-driver=convoy alpine sh
```

可以发现这时候的 -v 参数左边并不是一个宿主机目录，而是一个名称 vol_test，表示的是卷名称，如果没有 Convoy 会创建一个。Docker 会把这个数据卷挂载到容器中。如果启动失败请检查是否正确安装 Convoy。

Convoy 可以创建、删除、备份、还原数据卷。使用 Convoy 创建一个卷的命令如下：

```
$ sudo convoy create volume_name
```

删除一个卷：

```
$ sudo convoy rm volume_name
// or
$ sudo convoy delete volume_name
```

创建卷快照：

```
$ sudo convoy snapshot create vol_name --name vol_name_snap_1
```

备份一个卷：

```
$ sudo convoy backup create vol_name_snap_1 --dest vfs://opt/convoy
```

还原一个卷：

```
$ sudo convoy create res1 --backup vfs://opt/convoy
```

查看卷信息：

```
$ sudo convoy inspect volume_name
```

以上就是 Convoy 的基本用法，完整的使用说明可以查看 https://github.com/rancher/convoy/blob/master/docs/cli_reference.md。

9.4.3 Flocker 的使用

Flocker（Github 地址 <https://github.com/ClusterHQ/flocker>）是一个功能十分强大的卷插件，如图 9.1 是两者的 Logo，支持多种存储驱动，包括 OpenStack Cinder、EBS、ZFS 等。Flocker 不仅支持卷的共享，还支持卷的迁移。考虑到本书还未讲到集群的概念，下面的内容不会有关于 Flocker 多节点的使用方法。

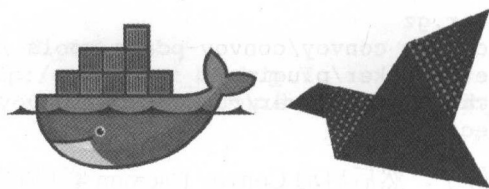


图 9.1 Docker 与 Flocker

以 Ubuntu 16.04 (64-bit) 为例安装 Flocker 工具。

```
$ sudo apt-get update
$ sudo apt-get -y install apt-transport-https software-properties-common
$ sudo add-apt-repository -y "deb https://clusterhq-archive.s3.amazonaws.com/ubuntu/${lsb_release --release --short}/${ARCH} /"
$ cat <<EOF > /tmp/apt-pref
Package: *
Pin: origin clusterhq-archive.s3.amazonaws.com
Pin-Priority: 700
EOF
$ sudo mv /tmp/apt-pref /etc/apt/preferences.d/buildbot-700
$ sudo apt-get update
$ sudo apt-get -y install --force-yes clusterhq-flocker-cli
$ systemctl enable flocker-control
$ systemctl start flocker-control
```

其他发行版的安装步骤在 <https://docs.clusterhq.com/en/latest/docker-integration/manual-install.html#installing-flocker>。

或者使用脚本安装也可以。

```
$ curl -sSL https://get.flocker.io/ | sh
```

这个脚本可以一键安装全部套件，包括 Flocker 的容器套件。

安装完成后，用户就可以使用 Flocker 来管理数据卷了。

```
$ docker run \
  --volume-driver flocker \
  -v volume_test:/container_vol \
  --name=container_test ...
```

9.5 本章小结

本章介绍了 Docker 的数据卷与数据卷容器，并对数据卷的挂载做出了详细说明，在这些内置功能的基础上，扩展了卷插件，介绍了卷插件的特性与使用方法。

对于 Docker 用户来说，数据管理是一个恒久的问题，希望本章能给读者一定的帮助。

第 10 章 网络管理

网络可以说是激活 Docker 体系的唯一途径，如果 Docker 没有相对出色的容器网络，则就没有现在的竞争力。

最初时，Docker 的网络解决方案并不理想，之后在 Docker 团队努力之下成绩不错，目前 Docker 网络虽然谈不上完善，但是无疑有了极大的进步，虽然 Docker 原生网络在性能上没有太大提升，但是相对更易用了。

激动人心的是，为了解决 Docker 大规模的集群部署，许多云计算服务商参与进来，大批 SDN 方案如雨后春笋冒出来，本章将从 Docker 原生网络讲起，然后深入介绍第三方的网络解决方案。

10.1 Docker 网络基础

Docker 目前对单节点的设备提供了映射容器端口到宿主机、容器互联两个网络服务。在集群部署上因为加入了 Swram，所以有了更好的网络支持。本节将讲解 Docker 的单节点网络基础内容。

10.1.1 端口映射

在 Docker 中容器默认是无法与外部通信的，需要在启动命令中加入对应参数才允许容器与外界通信。

当容器运行着一个 Web 服务时，需要把容器内的 Web 服务应用程序端口映射到本地宿主机的端口，这样用户通过访问宿主机指定端口时相当于访问容器内部的 Web 服务端口。

而这个参数就是 `-p`，在前面也有提到过这个参数，我们可以通过 `-P` 或者 `-p` 来指定端口映射。

当使用 `-P` 参数时，Docker 会随机映射一个端口至容器内部的开放端口，比如容器内部有两个端口开放，分别是 80 和 443，使用 `-P` 端口启动之后，Docker 会随机把 80 映射到宿主机的其中一个端口，同样 443 也是如此。

```
$ docker run -d -P --name test nginx:alpine
0c298d65c21a6b301dc1e59cd4f9ae2edcafb824691a2c388a77772df670091d
$ docker port test
443/tcp -> 0.0.0.0:32768
80/tcp -> 0.0.0.0:32769
```

使用 `docker port` 可以查看端口映射情况。如果使用 `docker logs` 可以查看到 Nginx 的日志：

```
$ docker logs test
172.17.0.1 - - [23/Oct/2016:14:31:34 +0000] "GET / HTTP/1.1" 200 612 "-"
"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.71 Safari/537.36" "-"

2016/10/23 14:31:35 [error] 7#7: *1 open() "/usr/share/nginx/html/
favicon.ico" failed (2: No such file or directory), client: 172.17.0.1,
server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "127.0.0.1:
32769", referer: "http://127.0.0.1:32769/"

172.17.0.1 - - [23/Oct/2016:14:31:35 +0000] "GET /favicon.ico HTTP/1.1" 404
571 "http://127.0.0.1:32769/" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36"
"-"
```

使用-P 参数不能自定义宿主机端口很不方便，因此通常都是使用-p 参数。-p（小写）可以指定要映射到本地的端口。支持的格式有：

```
Local_Port:Container_Port
Local_IP:Local_Port:Container_Port
Local_IP::Container_Port
```

上面 3 个格式意义不同，下面通过实验来诠释。

格式 1 (Local_Port:Container_Port)

```
$ docker run -d -p 8000:80 -p 4430:443 --name test nginx:alpine
6ca392c4abd3295144bb7f11f4f44a417a1a687badad79671db9fd525ed2db24
```

上面启动的是一个 Nginx 服务，在端口映射参数中指定了宿主机的 8000 端口映射到容器内部的 80 端口，同样宿主机 4430 端口映射到容器的 443 端口，如图 10.1 所示。可以多次使用-p 参数来添加多组映射关系。

格式 1 这种映射关系会映射所有接口地址，也就是说所有访客都可以通过访问宿主机的端口来访问容器服务。一般情况下都使用这种方式来映射端口。



图 10.1 指定宿主机端口

格式 2 (Local_IP:Local_Port:Container_Port)

与格式 1 不同，格式 2 可以映射到指定地址的指定端口，比如映射到 127.0.0.1 就会造成这个端口只能通过本机访问，外部无法访问这个容器服务，一般在测试等环境下会用到。同样的，如果指定了一个子网 IP，那么只有在同一个子网内的用户才可以访问，局域网外的用户无法访问。

```
$ docker run -d -p 127.0.0.1:8000:80 --name test nginx:alpine
6ca392c4abd3295144bb7f11f4f44a417a1a687badad79671db9fd525ed2db24
```

上面只能通过 127.0.0.1:8000 访问这个 Nginx 服务。

格式 3 (Local_IP::Container_Port)

该种格式有点像格式 2 与 -P 的结合，就是指定了哪些 IP 可以访问，但是宿主机端口却是随机分配映射的。

```
$ docker run -d -p 127.0.0.1::80 --name test nginx:alpine
8b16009b64f8351bc47aaldcbaba563c279c7ca9e39bd907066f39f9e1b5f059
$ docker port test
80/tcp -> 127.0.0.1:32768
```

格式 4 (指定传输协议)

上面 3 种格式还可以使用 tcp 或者 udp 标记来指定端口，例如：

```
$ docker run -d -p 80:80/tcp --name test nginx:alpine
$ docker run -d -p 80:80/udp --name test nginx:alpine
```

10.1.2 端口暴露

第 9 章曾讲到一个 EXPOSE 指令，当时说这个指令用于暴露端口，很多新手会误把端口暴露与端口映射混为一谈。

目前 Docker 有两种方式可以用来暴露端口：要么用 EXPOSE 指令在 Dockerfile 里定义，要么在 docker run 时指定 --expose=1234。这两种方式作用相同，但是，--expose 可以接受端口范围作为参数，比如 --expose=2000-3000。但是，EXPOSE 和 --expose 都不依赖于宿主机。默认状态下，这些规则并不会使这些端口可以通过宿主机来访问。

Dockerfile 的作者一般在包含 EXPOSE 规则时都只将其作为哪个端口提供哪个服务的提示，使用时，还要依赖于容器的操作人员进一步指定网络规则。

EXPOSE 或者 --expose 只是为其他命令提供所需信息的元数据，或者只是告诉容器操作人员有哪些已知选择。通过 EXPOSE 命令文档化端口的方式，有助于容器操作人员迅速确定服务启动命令。

在运行时暴露端口和通过 Dockerfile 的指令暴露端口两者没什么区别。在这两种方式启动的容器里，通过 docker inspect \$container_id | \$container_name 查看到的网络配置是一样的：

```
"NetworkSettings": {
  "PortMapping": null,
  "Ports": {
    "1234/tcp": null
  }
},
"Config": {
  "ExposedPorts": {
    "1234/tcp": {}
  }
}
```

可以看到端口被标示成已暴露，但是没有定义任何映射。注意一点，使用参数 --expose 是属于附加的，因此会在 Dockerfile 的 EXPOSE 指令定义的暴露端口之外添加新的端口。

10.1.3 容器互联

容器互联是除了端口映射外另一种可以与容器通信的方式。端口映射的用途是宿主机与容器的通信，而容器互联是容器之间的通信。

使用容器互联时，容器必须要有一个名字，也就是`--name`指定的值，这个值不管用户是否定义都会存在，默认由 Docker 随机生成词组。为了方便操作与记忆，一般手动设置容器名称：

```
$ docker run -d -p 8000:80 --name nginx nginx:alpine
```

如果忘记了设置名称，也可以通过 `docker rename` 来重命名容器。容器的名称是唯一的，如果该名字已经存在，新的同名容器会无法创建。

容器互联的参数是`--link`，这个参数在第4章曾经讲过，现在来看看如何让这两个容器连接起来。

下面先创建一个数据库容器（参数`-e MYSQL_ROOT_PASSWORD=password` 设置数据库 root 密码）。

```
$ docker run -d -e MYSQL_ROOT_PASSWORD=password --name db mysql:5.6
bd65de148e214d1b84bd07c094566b23370ffe927514b84d216a1eaf4c078e34
```

可以看到，上面启动的容器并没有在启动参数中添加任何端口映射参数，因此宿主机是无法访问容器内部的数据库的。

然后新建一个 Web 服务。

```
$ docker run -d -p 8000:80 --name php --link db:mysql abiosoft/caddy:php
3fc6ea829037b830e5e83ccc6ba56067606a4d14e5303539d69458c046c89513
```

这里`--link db:mysql` 使用参数值左右不同的写法来区分，左边表示的是刚才启动的容器名称，代表了 MySQL 容器，右边 `mysql` 表示的是后来运行的 PHP 容器。

启动完成后可以进入 PHP 容器，使用 `ping db` 查看是否连接成功。也可以新建一个 PHP 文件，测试是否能连接 MySQL 容器。

```
<?php
$con = mysql_connect("mysql","root","password");
if (!$con) {
    die('Could not connect: ' . mysql_error());
}
// some code
?>
```

注意上面的 `mysql_connect("mysql","root","password");` 之中没有使用 `localhost` 来连接，这是因为在容器中的 `/etc/hosts` 会实时根据环境而改变，例如使用 `--link` 连接容器的时候，会在容器的 `/etc/hosts` 文件中添加一行内容，使用 `docker exec` 进入容器查看即可知道：

```
$ docker exec -it php sh
/srv # cat /etc/hosts
127.0.0.1      localhost
::1           localhost ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters
172.17.0.9    mysql 76fa6e9ef2e5 db
```



```
172.17.0.10    0ce6458581f0
/srv #
```

上面加粗的内容已经证实了前面的说法，所以在容器互联操作上，数据库地址就是连接名称。

虽然--link 非常易于使用，能提供和端口映射几乎相同的功能。但--link 只能在容器之间使用。使用--link 参数创建的容器会使用容器的主机名和容器 ID 来更新自己的/etc/hosts 文件。

上面的例子中因为使用--link 时不会向宿主机映射端口，而是在容器之间建立一个通信隧道，因此避免了把数据库端口暴露到外部网络中。

值得注意的是，使用--link 时不只是允许容器之间通信，还包括环境变量的设置，例如上面的 PHP 容器，在连接到 db 容器的时候，实际上 PHP 容器的环境变量也发生了变化：

```
$ docker exec -it php env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=0ce6458581f0
MYSQL_PORT=tcp://172.17.0.9:3306
MYSQL_PORT_3306_TCP=tcp://172.17.0.9:3306
MYSQL_PORT_3306_TCP_ADDR=172.17.0.9
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_NAME=/php/mysql
MYSQL_ENV_MYSQL_ROOT_PASSWORD=password
MYSQL_ENV_GOSU_VERSION=1.7
MYSQL_ENV_MYSQL_MAJOR=5.6
MYSQL_ENV_MYSQL_VERSION=5.6.32-1debian8
HOME=/root
```

可以看到，使用 env 输出的内容实际上都是 db 容器的环境变量，但是通过--link 使得环境变量也交换了，这样做有助于迅速与数据库建立连接。如果 PHP 容器之前设置了 env，那么会被后来的 db 容器的 env 覆盖。

--link 参数在 docker run 中可以使用多次，例如上面的 PHP 容器允许连接多个数据库容器。

10.2 Docker 网络模式

在 Docker 刚出来的时候，网络方面一直是 Docker 的短板，为了解决这个问题，Docker 公司在 2015 年 3 月收购了 SocketPlane，并在一个月后发布了 Libnetwork，这是一个新的容器网络概念模型（Container Network Model），CNM 定义了标准的 API 用于容器网络配置，CNM 并不是网络实现，它是网络规范和网络体系，从研发的角度看就是一堆接口。

在 Linux 平台中，Libnetwork 通过 Network Namespace 机制实现隔离的，前面介绍过不同的 Namespace 拥有各自的网络设备、协议栈、路由表、防火墙规则等，Libnetwork 基于 Network Namespace 的灵活特性打造了 5 种 Docker 网络模式，本节将逐一讲解。

10.2.1 none 模式

none 模式表示不为容器配置任何网络功能，启用该模式只需要在启动时添加--net=none

即可。使用该命令启动的容器完全失去网络功能，即便设置了网络参数，例如：

```
$ docker run -d -p 8000:80 --name php --net=none abiosoft/caddy:php
```

此时打开 127.0.0.1:8000 是打不开页面的。可以通过 `docker exec` 查看容器内部的网络情况：

```
$ docker exec -it php ifconfig
lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1%32665/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

可以看到容器内部仅有一个 `lo` 环回接口，虽然容器目前没有网络功能，但是用户仍然可以手动为容器配置网络。以上面的容器为例：

首先为容器创建 `net` 命名空间：

```
$ PID=$(docker inspect -f '{{.State.Pid}}' mynetwork)
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$PID/ns/net /var/run/netns/$PID
```

然后创建一对 `veth` 接口 `A` 和 `B`，绑定 `A` 到自定义的网桥 `br0`（可以使用 `docker network create` 创建，或者默认使用 `docker0` 都可以）：

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif br0 A
$ sudo ip link set A up
```

最后将 `B` 放入容器中，命名为 `eth0`，启动并配置 `IP` 与默认网关：

```
$ sudo ip link set B netns $PID
$ sudo ip netns exec $PID ip link set dev B name eth0
$ sudo ip netns exec $PID ip link set eth0 up
$ sudo ip netns exec $PID ip addr add 10.10.10.25/24 dev eth0 // ip 与 br0
在同一网段中
$ sudo ip netns exec $PID ip route add default via 10.10.10.1
```

现在通过容器的 `ifconfig` 命令查看，情况如下：

```
$ docker exec -it php ifconfig
eth0    Link encap:Ethernet Wadded D2:27:3D:9F:E8:AA
        inet addr:10.10.10.25 Bcast:0.0.0.0 Mask:255.255.255.0
        inet6 addr: fe80::d027:3dff:fe9f:e8aa/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:8 errors:0 dropped:0 overruns:0 frame:0
        TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:648 (648.0 b) TX bytes:648 (648.0 b)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1%32665/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

10.2.2 container 模式

container 模式表示与另一个运行中的容器共享一个 Network Namespace，共享意味着拥有相同的网络视图。举个例子，以默认网络模式（bridge 模式）启动一个容器，并设置 HostName 和 DNS 如下：

```
$ docker run -itd --dns 8.8.8.8 -h testhost --name nginx nginx:alpine
8cbf147c3cfc8a67aedb6097c8elf93087a09e2fa158a74b4bcb4aa1133e2806
$ docker exec -it nginx ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:09
          inet addr:172.17.0.9  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:9%32757/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4670 (4.5 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1%32757/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

再启动一个容器，使用的是上面容器的网络。

```
$ docker run --rm --net=container:nginx -it nginx:alpine sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:09
          inet addr:172.17.0.9  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:9%32759/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:42 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:5522 (5.3 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1%32759/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

对比上面两个容器的 eth0 信息，可以发现网络配置完全相同，因为它们使用的是同一个 Network Namespace。

使用 cat /etc/hosts 可以看到两个容器的 hosts 文件都拥有同一个 hostname：

```
$ docker run --rm -h testhost --name testname nginx:alpine cat /etc/hosts
127.0.0.1      localhost
::1           localhost ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
```

```
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.9 testhost
```

10.2.3 host 模式

host 网络模式之前有过简单的介绍，它可以与主机共享 Root Network Namespace，容器有完整的权限操纵主机的网络配置，出于安全考虑，不推荐使用这种模式。

但是有时候必须使用这个模式，一个比较著名的情况就是 Eclipse 的 Che 项目，这是一个“未来版”的 Eclipse IDE，它基于 Docker 容器技术，为了有更好的体验，启动 Eclipse Che 时必须使用 host 模式。

启动 host 模式非常简单，依旧是在 `docker run` 中加入 `--net=host` 参数即可。例如：

```
$ docker run --rm --net=host -it nginx:alpine sh
/ # ifconfig
br-dlfc542ae048 Link encap:Ethernet HWaddr 02:42:C7:6B:D1:61
    inet addr:172.18.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
    inet6 addr: fe80::42:c7ff:fe6b:d161%32756/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:2509382 errors:0 dropped:0 overruns:0 frame:0
    TX packets:4196892 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:35107986621 (32.6 GiB) TX bytes:23178146920 (21.5 GiB)

docker0 Link encap:Ethernet HWaddr 02:42:67:41:81:6C
    inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
    inet6 addr: fe80::42:67ff:fe41:816c%32756/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:174864 errors:0 dropped:0 overruns:0 frame:0
    TX packets:301360 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:84093200 (80.1 MiB) TX bytes:96472408 (92.0 MiB)

enp3s0 Link encap:Ethernet HWaddr 24:F5:AA:BE:FF:56
    inet addr:172.16.168.168 Bcast:172.16.168.255 Mask:255.255.255.0
    inet6 addr: fe80::b3c9:63da:7ce4:d1d7%32756/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:4113743 errors:0 dropped:0 overruns:0 frame:0
    TX packets:8645566 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:4096
    RX bytes:1012120364 (965.2 MiB) TX bytes:12507192930 (11.6 GiB)

lo Link encap:Local Loopback
    inet addr:127.0.0.1 Mask:255.0.0.0
    inet6 addr: ::1%32756/128 Scope:Host
    UP LOOPBACK RUNNING MTU:65536 Metric:1
    RX packets:207753 errors:0 dropped:0 overruns:0 frame:0
    TX packets:207753 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1
    RX bytes:147644913 (140.8 MiB) TX bytes:147644913 (140.8 MiB)
```

这个容器使用了 host 模式，因此可以操作宿主机的网络配置，这是一件十分危险的事情，慎用。

10.2.4 bridge 模式

bridge 模式是 Docker 默认的网络模式，属于一种 NAT 网络模型，如图 10.2 所示。Docker daemon 在启动的时候就会建立一个 docker0 网桥（通过 -b 参数可以指定，后面会介绍到），每个容器使用 bridge 模式启动时，Docker 都会为容器创建一对虚拟网络接口（veth pair）设备，这对设备一端在容器的 Network Namespace，另一端在 docker0，这样就实现了容器与宿主机的通信（就像前面手动配置网络的例子一样）。

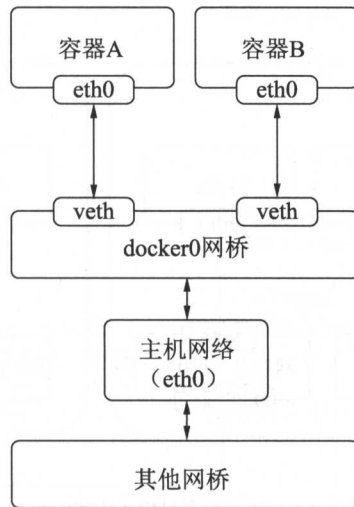


图 10.2 bridge 模式

在桥接模式下，Docker 容器与外部网络通信都是通过 iptables 规则控制的，这也是 Docker 网络性能低下的一个重要原因。使用 `iptables -vnL -t nat` 可以查看 NAT 表，在 Chain Docker 中可以看到容器桥接的规则。

使用桥接模式运行默认会分配一个子网 IP，例如：

```
$ docker run --rm -it nginx:alpine sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:09
          inet addr:172.17.0.9  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:9%32622/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8  errors:0  dropped:0  overruns:0  frame:0
          TX packets:5  errors:0  dropped:0  overruns:0  carrier:0
          collisions:0  txqueuelen:0
          RX bytes:938 (938.0 B)  TX bytes:418 (418.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1%32622/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0  errors:0  dropped:0  overruns:0  frame:0
          TX packets:0  errors:0  dropped:0  overruns:0  carrier:0
```

```
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

10.2.5 overlay 模式

overlay 模式是 Docker 原生的跨主机多子网网络模型，当创建一个新的网络时，Docker 会在主机创建一个 Network Namespace，Network Namespace 内有一个网桥，网桥上有一个 vxlan 接口，每个网络占用一个 vxlan ID，当容器被添加到网络中时，Docker 会分配一对 veth 网卡设备，与 bridge 模式类似，一端在容器里面，另一端在本地的 Network Namespace 上。

如图 10.3 所示，容器 A、B、C 都在主机 A 上面，而容器 D、E 则在主机 B 上面，现在通过 overlay 网络模型可以实现容器 A、B、D 处于同一个子网，而容器 C、E 处于另一个子网中。

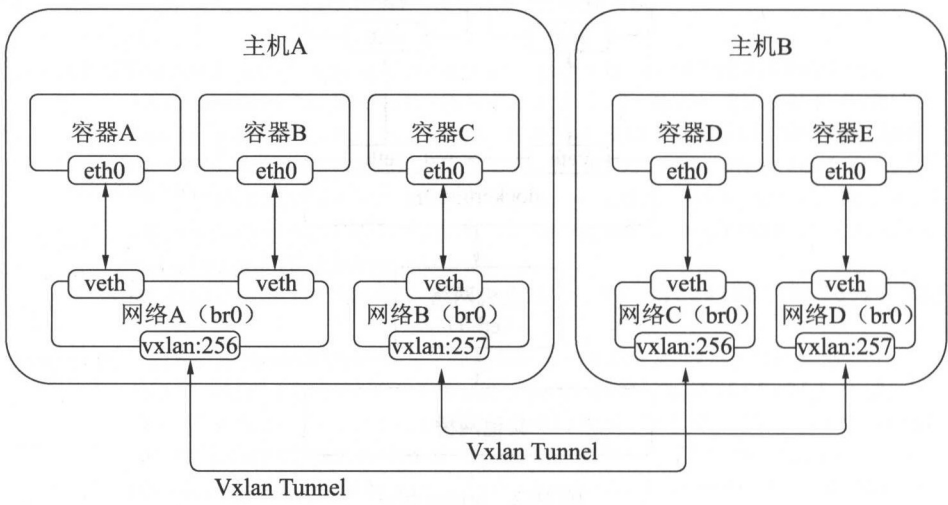


图 10.3 overlay 网络的结构示意图

Overlay 中有一个 vxlan ID，它的值范围为 256~1000 之间，vxlan 隧道会把每一个 ID 相同的网络沙盒连接起来实现一个子网。关于 overlay 网络模式的详细使用会在后面的章节中介绍，本节不做具体介绍。

10.3 Docker 网络配置

Docker daemon 在启动时可以设定网络参数。Docker clienet 在启动容器时也可以设定一些基本网络设定。最新的 Docker 中还添加了 network 的子命令。

10.3.1 Daemon 网络参数

在第 4 章的时候没有提到 docker daemon 命令，因为这个命令不属于 Docker Client 的一部分，它更像是一个 Service 运行在后端，本节会把 docker daemon 命令的网络参数拿出

来讲解，关于其他参数会在后面的相关章节介绍。

Docker daemon 的网络参数主要如下（仅选取部分作介绍）：

```
$ docker daemon --help
Usage: dockerd [OPTIONS]
```

A self-sufficient runtime for containers.

Options:

```
.....
-b, --bridge                Attach containers to a network bridge
// 指定 Docker daemon 启动时默认创建的网桥名称（默认为 docker0）
--bip                       Specify network bridge IP
// 指定 docker0 的 IP，注意不能与上面的 -b 一起用
.....
--default-gateway           Container default gateway IPv4 address
// 设置容器默认的 IPv4 网关
--default-gateway-v6       Container default gateway IPv6 address
// 设置容器默认的 IPv6 网关
.....
--dns=[]                   DNS server to use
// 设置容器默认的 DNS 地址
--dns-opt=[]               DNS options to use
--dns-search=[]            DNS search domains to use
.....
-H, --host=[]              Daemon socket(s) to connect to
// 指定 Docker client 与 Docker daemon 通信的 socket 地址，可以是 tcp 地址，也可以是 unix socket 地址，可以同时指定多个
.....
--icc=true                 Enable inter-container communication
// 设置是否允许容器间通信
.....
--ip=0.0.0.0               Default IP when binding container ports
// 容器端口暴露时绑定的主机 IP，一般默认即可
.....
--iptables=true            Enable addition of iptables rules
// 设置是否允许向 iptables 添加规则
--userland-proxy=true      Use userland proxy for loopback traffic
// 设置是否开启 docker-proxy，建议关掉，默认打开。用于端口映射时启动一个进程帮助数据转发，实际使用过程中这个功能有很多问题，例如长时间占用 CPU 资源，以及占用端口不释放导致容器启动失败等
```

Docker client 的网络参数主要是 docker run 的网络参数，在第 4 章中已经介绍过了。

10.3.2 配置 DNS

在讲解容器的章节中，我们提到一个“初始化层”，这个文件层实际上做了些什么呢？其实容器中的主机名和 DNS 配置信息都是通过 3 个系统配置文件来维护的，这些文件就放在“初始化层”，分别是/etc/hosts、/etc/resolv.conf、/etc/hostname。

启动一个容器的时候，在容器中使用 mount 命令可以查看这 3 个文件的挂载信息：

```
$ docker run --rm -it nginx:alpine sh
/ # mount
.....
/dev/sda1 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
/dev/sda1 on /etc/hostname type ext4 (rw,relatime,data=ordered)
```

```
/dev/sda1 on /etc/hosts type ext4 (rw,relatime,data=ordered)
.....
```

其中, `/etc/resolv.conf` 文件在创建容器的时候, 会默认与宿主机的 `/etc/resolv.conf` 保持一致, 而 `/etc/hosts` 中只会记录一些与容器相关的地址和名称信息, `/etc/hostname` 中记录的是主机名。

注意一点就是, 上面 3 个文件虽然在容器中允许修改, 但是当容器重启或者终止后就会丢失, 因此在容器中修改 `hosts` 时, 要在容器重启时在执行一次。而且这 3 个文件的修改不会被 `docker commit` 提交。

`Docker client` 可以通过 `-h` 修改 `hostname`, `--dns` 修改 DNS 地址等, 这些信息会在容器启动时写入上述 3 个文件。

10.3.3 network 命令

使用 `docker network ls` 命令可以查看当前主机的网桥情况。

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
be9c5bf8ae27        bridge             bridge              local
fbd9a977aa03        host               host                local
d1fc542ae048        nginx_default      bridge              local
e81af24d643d        none              null                local
```

删除一个网络可以使用 `docker network rm` 命令, 创建一个网络可以使用 `docker network create` 命令。

```
$ docker network create foo
a01ac70cd9edfee5008452f9b4eb927c609418f9104cc6590adff724aa1532df
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
be9c5bf8ae27        bridge             bridge              local
a01ac70cd9ed        foo                bridge              local
fbd9a977aa03        host               host                local
d1fc542ae048        nginx_default      bridge              local
e81af24d643d        none              null                local
$ docker network rm foo
Foo
```

默认情况下, 新建网络默认使用桥接模式, 如果想要新建其他模式的网络可以使用 `-d` 指定 (指定 `overlay` 模式需要服务发现组件运行正常, 后面章节会具体介绍)。

10.4 本章小结

网络可以说是 `Docker` 中最复杂的一部分, 本章主要讲述了单节点的容器网络知识, 包括端口映射、端口暴露、容器互联、网络模式等, 学习完本章可以说 `Docker` 已经完全入门了。

如果说前面都是 `Docker` 的操作基础和基本应用, 从本章开始, 后面的章节都是需要有一定基础才能理解的内容了。`Docker` 网络一直以来都属于 `Docker` 比较薄弱的一环, 因此目前的解决方案算不上十分完善, 因为不完善导致的一些问题也会影响用户的学习与使用, 如果读者遇到问题, 要充分利用搜索引擎自己尝试解决。

第3篇

Docker 进阶实战

- » 第11章 操作系统
- » 第12章 编排工具 Compose
- » 第13章 Web 服务器与应用
- » 第14章 数据库
- » 第15章 编程语言
- » 第16章 Docker API 介绍
- » 第17章 私有仓库
- » 第18章 集群网络
- » 第19章 Docker 安全

第 11 章 操作系统

Docker 镜像一般都是基于某个系统镜像构建而来，这个操作系统旨在为 Docker 容器提供环境支持，所以选用适合的操作系统作为镜像的基础是容器稳定运行的一道保障。

本章的目的是了解那些适合在镜像底层工作的操作系统，使大家在构建自己的镜像时更好地选择适合自己的基础镜像。

本章主要包括三部分：

- 了解 Docker 镜像的基础镜像。
- 尝试从零开始构建基础镜像。
- 认识为 Docker 而设计或者改进的操作系统。

11.1 Alpine 发行版

Alpine Linux 是一个社区开发的面向安全应用的轻量级 Linux 发行版。不同于通常 Linux 发行版，Alpine Linux 采用了 musl libc 和 busybox 以减小系统的体积和运行时资源消耗。此外，Alpine Linux 还提供了自己的包管理工具 apk，这使得基于 Alpine Linux 构建的镜像体积都非常小，资源消耗也很少。

值得注意的是，Alpine Linux 的创始人 Natanael Copa 已经加入 Docker，Docker 公司正在尝试把 Alpine Linux 作为 Docker 默认基础镜像。

11.1.1 官方镜像

Alpine Linux 的官方镜像可以通过 `docker pull alpine` 命令来获取最新版本，可以看到最新版本的 Alpine Linux 只有 4.797 MB。

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine

d0ca440e8637: Pull complete
Digest: sha256:f655166f57d91bdfc8b3bc75a20391b7516de9f48ca761249c185fcb022124d2
Status: Downloaded newer image for alpine:latest
$ docker images
alpine          latest          13e1761bf172    10 days ago     4.797 MB
```

如果需要 Alpine Linux 的某个特定版本，只需要加上版本号即可，如 `docker pull alpine:edge` 则是拉取 Alpine Linux 的 edge 版本。在 <https://hub.docker.com/r/library/alpine/tags/> 上，可以找到 Alpine Linux 的全部版本。

也可以在终端下使用搜索命令寻找需要的第三方镜像，搜索结果中官方镜像会有 official 标记，如图 11.1 所示。

```
$ docker search alpine
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
alpine	A minimal Docker image based on Alpine Lin...	1480	[OK]	
anapsix/alpine-java	Oracle Java 8 (and 7) with GLIBC 2.23 over...	151		[OK]
frolvlad/alpine-glibc	Alpine Docker image with glibc (~12MB)	43		[OK]
container4armhf/armhf-alpine	Automatically built base images of Alpine ...	32		[OK]
mhart/alpine-node-auto	Automated build of mhart/alpine-node - a...	30		[OK]
zzrozt/alpine-caddy	Caddy Server Docker Container running on A...	23		[OK]
davidcaste/alpine-tomcat	Apache Tomcat 7/8 using Oracle Java 7/8 wi...	10		[OK]
chickenzord/alpine-gradle	Docker image for Gradle based on Alpine Linux	5		[OK]
nimmis/alpine-java	This is docker images of Alpine 3.3 with d...	5		[OK]
orax/alpine-armhf	Daily built Alpine-Linux Docker image for ...	5		[OK]
nimmis/alpine-micro	A very small container (8.4 Mb) based on A...	4		[OK]
lscience/alpine	Base Docker images based on Alpine Linux	4		[OK]
webhippie/alpine	Docker images for alpine	3		[OK]
pandada8/alpine-python	An alpine based python image	3		[OK]
inzinger/alpine-ruby	Minimal Image with Ruby based on Alpine.	3		[OK]
bluebu/rails-alpine	rails-alpine with different ruby version	3		[OK]

图 11.1 在终端中搜索 Alpine

11.1.2 运行 Alpine Linux

现在尝试启动一个 Alpine Linux 容器，并进入容器环境内部。

```
$ docker run --rm -it alpine sh
/ #
```

执行后，终端会进入 Alpine Linux 容器内部。可以进入/bin 目录下查看 Alpine Linux 提供了哪些基本工具。

```
/ # cd bin
/bin # ls
ash      df        gunzip    mknod     ps        sync
base64   dmesg     gzip      mktemp    pwd       tar
bbconfig dnsdomainname hostname  more      reformime touch
busybox  dumpkmap  ionice    mount     rev       true
cat       echo      iostat    mountpoint rm         umount
catv     ed         ipcalc    mpstat    rmdir     uname
chgrp    egrep     kbd_mode  mv         run-parts usleep
chmod    false     kill      netstat   sed        watch
chown    fatattr   ln         nice      setserial zcat
conspy   fdflush   login     pidof     sh
cp        fgrep     ls         ping      sleep
cpio     fsync     lzop      ping6     stat
date     getopt   makemime  pipe_progress stty
dd        grep      mkdir     printenv  su
/bin #
```

在 Alpine 中内置了 Busybox 的工具包，BusyBox 有的 Alpine 基本都有。BusyBox 是一个比 Alpine 更小的 Linux 系统，11.1.3 节会具体介绍。

11.1.3 构建基于 Alpine Linux 的镜像

现在来点好玩的，尝试构建一个基于 Alpine Linux 的镜像。我们以 MySQL Client 为例，所使用的 Dockerfile 如下：


```
FROM alpine:3.1
RUN apk add --update mysql-client && rm -rf /var/cache/apk/*
ENTRYPOINT ["mysql"]
```

然后在 Dockerfile 目录下执行构建命令，构建镜像。

```
$ docker build -t mysql-client .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM alpine
---> 13e1761bf172
Step 2 : RUN apk add --update mysql-client && rm -rf /var/cache/apk/*
---> Running in b5c2abb9ac4b
..... 此处省略部分内容 .....
OK: 39 MiB in 17 packages
---> 01a3ed87660c
Removing intermediate container b5c2abb9ac4b
Step 3 : ENTRYPOINT mysql
---> Running in 6ef58d7686a6
---> 4a75cd2e5a5f
Removing intermediate container 6ef58d7686a6
Successfully built 4a75cd2e5a5f
```

构建完成后可以看到刚才的镜像只有 35MB 而已，和使用 Ubuntu 构建的同样功能镜像相比小了将近 10 倍（构建镜像相关知识详见第 5 章和第 6 章）。

11.1.4 Alpine Linux 软件仓库

前面提到，Alpine Linux 提供了自己的包管理工具 apk。但是对于软件仓库内有哪些软件，就需要去 Alpine Linux 网站搜索了，如果没有需要的软件，就需要自己构建软件包，不过那些知识不是本书的范围，有兴趣的读者可以了解下。

Alpine Linux 仓库地址主要有两个，一个是官方仓库 <http://nl.alpinelinux.org/alpine/edge/main>。

另一个是社区仓库 <http://nl.alpinelinux.org/alpine/edge/community>。

也可以在官网上直接搜索 <https://pkgs.alpinelinux.org/packages>。

现在可以在 Dockerfile 中添加软件仓库的软件源，这里以 Node.js 为例，添加社区仓库的 nodejs 只需要在后面加上 @community。

```
FROM alpine
RUN echo '@edge http://nl.alpinelinux.org/alpine/edge/main' >> /etc/apk/repositories
RUN echo '@community http://nl.alpinelinux.org/alpine/edge/community' >> /etc/apk/repositories
RUN apk update && apk upgrade \
    && apk add ca-certificates nodejs@community \
    && npm uninstall -g npm \
    && rm -rf /var/cache/apk/*
```

Alpine Linux 镜像虽然小巧，但是提供了常见的 Linux 命令，并且其软件仓库也提供了丰富的软件扩展。

11.2 Busybox 发行版

Busybox 是一个集成了一百多个最常用 Linux 命令和工具的软件工具箱，它在单一的可执行文件中提供了精简的 UNIX 工具集，堪称 Linux 工具里的“瑞士军刀”。Busybox 比 Alpine Linux 体积更小，但是其并没有包管理工具，所以在扩展上稍微麻烦一些。

11.2.1 官方镜像

Busybox 镜像可以通过 `docker pull busybox` 拉取最新版本。

```
$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:4a887a2326ec9e0fa90cce7b4764b0e627b5d6afcb81a3f73c85dc29cea00048
Status: Downloaded newer image for busybox:latest
```

使用 `docker search busybox` 搜索官方仓库里面全部的 Busybox 镜像，查看镜像可以看到大小只有 1.113MB，比 Alpine Linux 还小。

```
$ docker images
busybox          latest          47bcc53f74dc    8 weeks ago     1.113 MB
```

11.2.2 运行 Busybox

启动一个 Busybox 容器，并进入容器内部。

```
$ docker run --rm -it busybox sh
/ #
```

执行后，终端会进入 Busybox 容器内部。可以进入 `/bin` 目录下查看 Busybox 提供了哪些基本工具。

```
/ # cd /bin && ls
```

可以看到 Busybox 提供了很多基本工具（比 Alpine 更丰富的内置工具），不愧为“瑞士军刀”。

11.2.3 构建基于 Busybox 的镜像

虽然 Busybox 很小，但是扩展起来很麻烦，一般用来运行一些编译好的二进制文件。

```
FROM busybox
COPY ./my-static-binary /my-static-binary
CMD ["/my-static-binary"]
```

Busybox 的应用场景需要比较少，看个人需求。相比之下，更推荐读者使用 Alpine Linux 作为基础镜像。

11.3 Debian/Ubuntu 发行版

作为 Linux 上的一个重要分支, Debian / Ubuntu 可谓是无人不晓,加之 Debian / Ubuntu 社区都非常活跃, Docker 目前也是把 Debian 和 Ubuntu 作为基础镜像的默认镜像,用来构建常用工具的镜像,这些镜像都会托管在 hub.docker.com/u/library 中。

11.3.1 官方镜像

通过 `docker search debian` 搜索 Debian 镜像,因为 Debian 以及 Ubuntu 镜像实在太多,搜索时可以通过收藏数限制显示数量,如搜索 Ubuntu,使用 `docker search -s 10 ubuntu`,这样就只显示被收藏 10 次以上的镜像了。

官方镜像则直接使用 `docker pull debian` 或者 `docker pull ubuntu` 即可拉取最新版本。值得注意的是,不推荐读者使用 Debian/Ubuntu 的最新版本,而是使用可信赖的标签,如 Debian 的 `jessie` 或者 `wheezy` 标签,同样,Ubuntu 也有其可信赖标签 `trusty`、`wily`、`xenial` 等。

```
$ docker pull ubuntu:trusty
$ docker pull debian:jessie
```

11.3.2 运行 Debian/Ubuntu

下面就以 Ubuntu 14.04 为例,让我们进入一个 Docker 版的 Ubuntu 操作系统里面体验一番吧。

使用 `-it` 参数进入并启动交互 `bash`, 参数 `--rm` 可以在退出容器时直接删除容器:

```
$ docker run -it --rm ubuntu:trusty bash
root@14d811787f68:/#
```

进入容器后,会以 `root` 账号登录到容器内部,上面的例子中 `14d811787f68` 是容器的 ID,需要注意的是,这里的 `root` 账号并非和服务器的 `root` 账号,而是一个容器 `root` 账号,对于服务器来说这个 `root` 账号只是一个普通用户。这个 `root` 账号只可以在容器内部以 `root` 身份工作(这里指一般情况,事实上容器也可以获得服务器本地的最高权限,但需要相应的参数与配置,详见第 5 章)。

接下来如果尝试在容器内部安装软件(这里以安装 `curl` 为例),会发现错误提示:

```
root@14d811787f68:~# apt install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package curl
```

这是因为 Docker 镜像为了精简镜像体积,默认删除了一些东西,只需要执行一次更新即可。

```
root@14d811787f68:~# apt update
Ign http://archive.ubuntu.com trusty InRelease
Get:1 http://archive.ubuntu.com trusty-updates InRelease [65.9 kB]
```

..... 此处省略部分内容

```
Get:22 http://archive.ubuntu.com trusty/universe amd64 Packages [7589 kB]
Fetched 21.8 MB in 8s (2717 kB/s)
Reading package lists... Done
```

更新之后就可以使用 `apt install curl` 安装刚才的应用了。接下来安装一个 Nginx 服务，然后使用 `curl` 来测试访问该 Web 服务。

```
root@14d811787f68:~# apt install nginx
Reading package lists... Done
Building dependency tree
Reading state information... Done
..... 此处省略部分内容 .....
```

接下来启动这个 Nginx 服务，然后使用 `curl` 来测试本地访问：

```
root@14d811787f68:~# service nginx start
root@14d811787f68:~# service nginx status
* nginx is running
root@14d811787f68:~# curl 127.0.0.1
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
..... 此处省略部分内容 .....
```

可以看到 Nginx 服务正常运行。当然这样的 Web 服务在外界是访问不了的，因为使用的是 `-it` 参数启动，如果想外部设备访问，则需要使用前面提到的 `-p` 参数来指定映射端口。

同时，也不推荐读者使用这种方式部署和运行 Web 服务应用，因为这样违反了 Docker 的理念，容器并不是虚拟机。如何使用 Docker 部署 Web 服务，将是第 12 章的内容。

11.3.3 构建基于 Debian/Ubuntu 的镜像

使用 Debian / Ubuntu 的镜像非常多，这里以 Ubuntu 为基础镜像，构建一个 Python 开发环境为例。

```
FROM ubuntu:trusty
RUN apt update && apt install -y python-dev
.....这里省略一部分，关键是下面.....
RUN apt-get autoremove -y && \
apt-get autoclean -y && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

该例想说明的并不是如何构建一个镜像，而是希望读者在构建镜像时注意对镜像的清理，保持镜像体积轻巧的特点，不要给镜像增加不必要的文件。事实上在前面章节说过，Dockerfile 很像一份自动化的 Linux 命令集合，选择 Debian/Ubuntu 作为基础镜像，意味着用户可以使用最熟悉的工具构建自己的镜像，避免因使用不熟悉的包管理工具造成不必要的麻烦。

11.4 CentOS/Fedora 发行版

CentOS 和 Fedora 都是基于 Redhat 的 Linux 发行版。前者以兼容 Redhat 软件以及稳定的运行保障而著称，后者则以新特性、新技术而著称，更多地体现实验性，主要面向个人用户。

11.4.1 官方镜像

CentOS 的官方镜像托管地址在 Docker Hub 的官方仓库中，地址为 https://hub.docker.com/_/centos/。

使用 `docker pull centos` 可以拉取 CentOS 的 latest 标签的镜像（latest 标签的 CentOS 版本为 7.2，使用时注意版本号的区别，每个大版本的 CentOS 都有不小改变）。如果用户不需要或者想寻找第三方的 CentOS 镜像，可以使用命令 `docker search centos` 搜索 CentOS，或者在 Docker Hub 网页中搜索相关资源。为保证用户的数据安全，建议选择可靠的基础镜像来源，对于不明来历的镜像请谨慎使用。

同样的，Fedora 的官方镜像也托管在 Docker Hub 中，地址是 https://hub.docker.com/_/fedora/。

使用 `docker pull fedora` 可以拉取 Fedora 的 latest 标签的镜像，latest 标签的镜像中 Fedora 的版本为 25。类似地使用 `docker search fedora` 搜索第三方 Fedora 镜像。仓库中还额外提供了一个 rawhide 的标签，Rawhide 是 Fedora 开发版本的代号。这个版本包含了一个叫做 rawhide 的源并且包含所有每天最新构建的 Fedora 软件包。

11.4.2 运行 CentOS/Fedora

基础镜像的使用与普通镜像并无区别，CentOS / Fedora 在操作上与其他 Linux 操作系统也并没有太大差别，包管理工具虽不相同，但是软件用法几乎一致。

例如给 CentOS 基础镜像安装基本的网络工具：

```
$ docker run --rm -it centos bash
[root@d57af7200a47 bin]# yum install -y net-tools
Loaded plugins: fastestmirror, ovl
.....
Installed:
  net-tools.x86_64 0:2.0-0.17.20131004git.el7
Complete!
```

然后可以使用 `ifconfig` 等命令更方便地查看网络相关的信息。CentOS 的基础镜像去掉了很多常见的组件，使用过程中需要用户手动添加。

与 CentOS 不同的是，Fedora 在基础镜像中重定向了 `yum` 包管理工具到 `dnf` 中，Fedora 希望用户使用 `dnf` 包管理工具而不是 `yum`，因此在使用 Fedora 镜像时请使用 `dnf` 安装软件：

```
$ docker run --rm -it fedora bash
[root@cf415ed4aa40 /]# dnf install -y net-tools
```

```

Fedora 25 - x86_64
2.2 MB/s | 50 MB      00:22
Fedora 25 - x86_64 - Updates
5.6 MB/s | 16 MB      00:02
Last metadata expiration check: 0:00:07 ago on Tue Jan 17 12:48:40 2017.
Dependencies resolved.
.....
Installed:
  net-tools.x86_64 2.0-0.38.20160329git.fc25
Complete!

```

因为 Fedora 的本地仓库缓存比 CentOS 大很多，构建镜像时勿忘清理本地软件仓库缓存，否则这也是增加镜像或容器体积的一个因素。

11.5 CoreOS 发行版

CoreOS 是一个基于 Linux 内核的轻量级操作系统，为了计算机集群的基础设施建设而生，专注于自动化，轻松部署，安全，可靠，规模化。与其他通用 Linux 发行版（Ubuntu、Debian、Redhat）相比，它具有体型小，消耗小，支持滚动更新等特点。

除此之外，作为一个操作系统，CoreOS 提供了在应用容器内部署应用所需要的基础功能环境以及一系列用于服务发现和配置共享的内建工具，内置的分布式系统服务组件给开发者和运维者组建分布式集群、部署分布式服务应用带来了极大便利。

需要注意的是，CoreOS 并不属于 Docker 基础镜像的一种，这不是构建镜像用的系统，而是运行 Docker（准确来说是容器）的系统。

11.5.1 为什么使用 CoreOS

本书的第一部分已经介绍了 Docker 的大部分功能，我们注意到，绝大部分进入 Docker 浪潮的公司都在走 Pass 的道路，通过自定义一个云平台然后提供服务，包括 Docker 的母公司，这也是一条最正统的大道。

但也有一部分人注意到，与其建造平台不如把平台做成一个操作系统，CoreOS 走的就是这样一条路，把操作系统与 Docker 相结合，不必要的东西全部删除。

在 CoreOS 世界里，需要服务发现，那就集成 etcd，需要管理服务，就集成 system，需要编排部署，就集成 Fleet。这样，在操作系统层面下，负责构建容器云的工程师就不需要自建一套体系。

在 CoreOS 的世界里还有很多有趣的细节值得学习，但不是本章的内容，在 Docker 生态部分会再次接触 CoreOS。

11.5.2 用 Vagrant 安装 CoreOS

CoreOS 的官网是 <https://coreos.com/>，截止 2016 年 5 月，国内外主流云平台提供商如 Amazon EC2、Google Compute Engine、Microsoft Azure、Digital Ocean 等均提供了 CoreOS 镜像，通过这些服务，可以一键建立一个 CoreOS 实例，这似乎也是 CoreOS 官方推荐的主

流安装方式。

CoreOS 当然也支持其他方式的安装，如支持虚拟机安装（Vagrant + Virtualbox）、PXE（Preboot execute environmen）安装以及 ISO 安装到物理硬盘等方式。如图 11.2 是 CoreOS 官网提供的诸多安装方式，供用户选择。



Bare Metal
PXE, iPXE, install to Disk

Cloud Providers
EC2, DigitalOcean, GCE, Rackspace, Azure,
Brightbox

Virtualization Platforms
Vagrant, VMware, QEMU, Openstack, Eucalyptus,
ISO Image

图 11.2 CoreOS 官方支持的安装方式

这里使用 Vagrant 安装 CoreOS 为例，因为在云服务器环境中一般不支持自定义操作系统，所以不介绍裸机安装 CoreOS，有兴趣的读者可以在官网找到相关资料。

通过 core-vagrant 安装的直接结果是 CoreOS 被安装到一个 VirtualBox 虚拟机中，之后利用 Vagrant 命令来进行 CoreOS 虚拟机的启停。

在安装 CoreOS 之前，需要确保以下软件已经安装到系统中。

- VirtualBox，官网下载地址为 <https://www.virtualbox.org/wiki/Downloads>。
- Vagrant，官网下载地址为 <http://www.vagrantup.com/downloads.html>。
- Git，直接使用包管理工具 install git 安装即可。

在上面的软件全部安装完成之后，就可以开始部署 CoreOS 了。

```
$ git clone https://github.com/coreos/coreos-vagrant/
$ cd coreos-vagrant
$ vagrant up
$ vagrant ssh
```

执行完毕 vagrant ssh，会自动生成 SSH 的一些信息，使用熟悉的 SSH 终端工具连接即可：

```
Host: 127.0.0.1
Port: 2222
Username: core
Private key: /path/.vagrant.d/insecure_private_key
// 这个路径下可以找到 SSH 登录的密钥
```

进入 CoreOS 后，就可以直接使用 Docker 了，虽然成功安装了 CoreOS，但在实际应用中，CoreOS 多以 Cluster 形式呈现，也就是说要启动多个 CoreOS 实例。

接下来通过修改配置文件来启动多个 CoreOS 实例。

```
$ mv config.rb.sample config.rb
$ mv user-data.sample user-data
```

修改 config.rb 里面的实例数量：

```
$ num_instances=1
// 如果安装单个 CoreOS 就写 1，如果是集群就写大于 1 的数字
```

该配置文件中的数据会覆盖 Vagrantfile 中的默认配置。

3 个实例中的 Etcd2 要想组成集群还需要一个配置修改，那就是在 etcd.io 上申请一个 token:

```
$ curl https://discovery.etcd.io/new
https://discovery.etcd.io/fe79223607223aae273dc5f233eb249a
```

将这个 token 配置到 user-data 中的 Etcd2 下:

```
etcd2:
  #generate a new token for each unique cluster from https://discovery.
  etcd.io/new
  #discovery: https://discovery.etcd.io/<token>
  discovery: https://discovery.etcd.io/fe79223607223aae273dc5f233eb249a
```

之后的步骤和前面一样，使用 Vagrant 启动。

```
$ vagrant up
Bringing machine 'core-01' up with 'virtualbox' provider...
Bringing machine 'core-02' up with 'virtualbox' provider...
Bringing machine 'core-03' up with 'virtualbox' provider...
==> core-01: Checking if box 'coreos-stable' is up to date...
.....省略部分内容.....
==> core-03: Running provisioner: shell...
    core-03: Running: inline script
$ vagrant ssh core-02
// 连接其中一个实例
```

虽然现在安装好了，但 CoreOS 的各种服务组件的功用、配置、如何与 Docker 配合形成分布式服务系统、如何用 Google Kubernetes 管理容器集群等都需要进一步学习，这也是 Docker 生态部分的重点内容。

因为国内网络情况较为特殊，教程中的环境为国外服务器，如果读者服务器地理位置为国内，则需要自行解决网络问题，否则会报错，对于没有办法解决网络问题的读者，推荐使用国内的镜像源，如阿里云镜像、网易镜像都有 CoreOS 的镜像。

11.6 RancherOS 发行版

RancherOS 是 Rancher Labs 的一个开源项目，旨在提供一种在生产环境中大规模运行 Docker 的最小最简单的方式。它只包含运行 Docker 必须的软件，其二进制下载包只有约 20MB。

与 CoreOS 一样，RancherOS 也不属于 Docker 基础镜像的一种，而是运行 Docker 的系统。

11.6.1 为什么使用 RancherOS

如图 11.3 所示，在 RancherOS 中，一切都是由 Docker 管理的容器。RancherOS 会启动两个 Docker 实例。一个称为系统 Docker，是内核启动的第一个进程，即 PID 1。它取代了其他 Linux 发行版本中的初始化系统，如 sysvinit 或 system，负责初始化系统服务，如 udev、DHCP 和控制台，并将所有系统服务作为 Docker 容器进行管理。

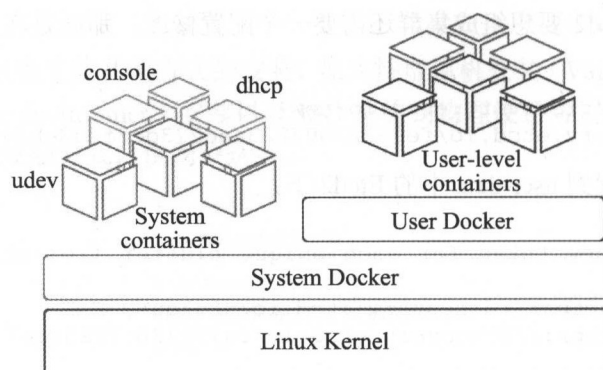


图 11.3 RancherOS 的结构

系统 Docker 会创建一个特殊的系统服务容器，即用户 Docker，主要负责创建容器。所有的用户容器都运行在用户 Docker 容器中，因此删除所有的用户容器并不会影响运行 RancherOS 服务的系统容器。

11.6.2 在服务器安装 RancherOS

如果已经安装了 Docker Machine，那么直接执行安装命令即可。

```
$ docker-machine create -d virtualbox \
--virtualbox-boot2docker-url
https://releases.rancher.com/os/latest/rancheros.iso \
<MACHINE-NAME>
```

如果打算安装到硬盘，那么可以从 Github 上下载最新版本的 RancherOS，地址为 <https://github.com/rancher/os/releases>。

然后刻录：

```
$ sudo mkfs.ext4 -L RANCHER_STATE /dev/sda
```

重启后，RancherOS 将会从 sda 启动。RancherOS 带有一个简单的安装程序，安装需要先配置 cloud-config.yml，具体配置方法在官网写得非常完善 <http://docs.rancher.com/os/cloud-config/>。

本书以默认的 cloud-config.yml 为例，启动安装程序。

```
$ sudo ros install -c cloud-config.yml -d /dev/sda
INFO[0000] No install type specified...defaulting to generic
Installing from rancher/os:v0.4.5
Continue [y/N]:
```

输入 y 然后回车。

```
Unable to find image 'rancher/os:v0.4.5' locally
v0.4.5: Pulling from rancher/os
... ..
Status: Downloaded newer image for rancher/os:v0.4.5
+ DEVICE=/dev/sda
... ..
+ umount /mnt/new_img
Continue with reboot [y/N]:
```

输入 y 重启之后可以使用 SSH 连接到 RancherOS 了, 用户名和密码默认都是 rancher。

```
$ ssh -i /path/to/private/key rancher@<ip-address>
```

注意, 一定要配置 cloud-config.yml 文件, 否则安装完之后自己就登录不上了。一定要注意安装硬盘的位置, 不要装错地方了。另外, RancherOS 默认 NS 服务器是 Google 的, 国内环境需要自己做调整, 在 cloud-config.yml 里面修改或者修改配置文件/etc/resolv.conf 都可以解决这个问题。

因为 RancherOS 项目比较“年轻”, 目前 RancherOS 只支持国外数家云服务提供商, 国内基本还没有云服务商支持 RancherOS。

11.6.3 基于 RancherOS 的 Docker 管理

在说 Docker 管理之前, 先来看看 RancherOS 和前面提到的 CoreOS 的区别。RancherOS 系统启动非常快, 里面只有两个关键部分, 一个是 System-Docker, 另外一个为用户 Docker。和 CoreOS 使用 systemd 管理容器不同, RancherOS 直接把一切都作为 Docker 容器, 系统的 Docker 运行了所有系统中需要的进程。

从图 11.4 中可以看到 RancherOS 比 CoreOS 更激进与轻量, 但同时 RancherOS 也缺少了一些主流的集群管理工具, 需要工程师们自行搭建。

RancherOS 提供了一套管理自身的工具 ros, 通过 ros 可以完成对操作系统本身的大部分操作, 包括升级、配置、安装。使用 -h 参数可以查看其具体用法:

```
$ sudo ros -h
```

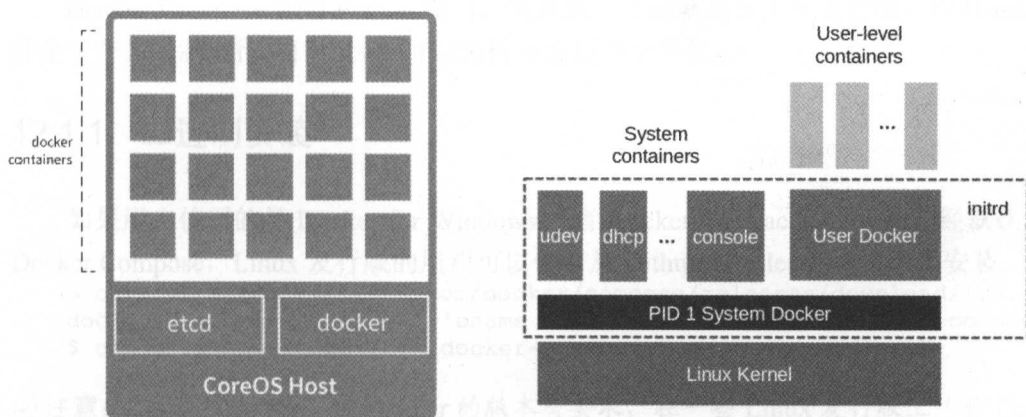


图 11.4 CoreOS 和 RancherOS 的区别

11.7 本章小结

本章前面 4 节带领读者了解了 Docker 镜像的基础镜像, 并逐一尝试构建自己的基础镜像, 11.5 和 11.6 节则介绍了为 Docker 而设计或者改进的操作系统。

除了本章中介绍到的镜像, 还有很多第三方开发者上传到 Docker Hub 上的系统镜像,

读者可以根据自己的需要拉取使用。

一般来说，官方的镜像体积都比较小巧，只安装了基本的依赖。一个精简的系统有利于安全、稳定和高效地运行。当然，也不乏“高手在民间”的情况，一些个人上传的镜像质量也非常之高，读者需要自己阅读镜像的 `Dockerfile` 来判断镜像的质量。

此外，如果想查看下载镜像的详细信息，可以通过 `docker inspect imagesID` 来查看更多信息。

最后和读者一起回顾了第5章和第6章的一些知识点，同时也有不少问题被留到了后面的章节做解答。第12章开始将在本章基础镜像的基础上，部署一些有实用性的 Web 服务应用。

第 12 章 编排工具 Compose

还记得使用 `--link` 连接容器的内容吗？我们需要先启动数据库容器，然后再启动应用容器，最后可能还要启动反代理容器，这样才算完整部署一个 Web 应用。这需要使用三句命令才能部署，操作起来很麻烦，而且不能把三个容器统一起来管理，就连三句命令都要自己动手保存起来，很是麻烦，那么有没有什么工具可以统一管理多个互相关联的容器呢？

当然是有的，这就是本章的主角——Docker Compose，Docker Compose 原本是 Docker 社区的一个基于 Python 编写的容器编排工具，后来被 Docker 项目组合并，改名为 Docker Compose。

简单来说，Docker Compose 是一个用来组装管理多容器应用的工具，它可以根据配置文件自动构建、管理、编排一组容器，极大方便了用户对多容器应用的操作。


12.1 安装 Docker Compose

Docker Compose 使用 Python 编写，因此从一开始就是全平台支持的，而且 release 文件是一个二进制执行文件，因此可以轻松安装到各个平台。

12.1.1 二进制安装

如果用户使用的是 Docker for Windows 或者 Docker for macOS，那么已经默认安装了 Docker Compose，Linux 发行版的用户可以使用从 Github 的 release 页面下载安装：

```
$ curl -L https://github.com/docker/compose/releases/download/1.9.0-rc1/  
docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose  
$ chmod +x /usr/local/bin/docker-compose
```

 **注意：**Docker Compose 对 Docker 的版本有要求，在一些 Linux 发行版上（如 CentOS 6 等）是无法使用 Docker Compose 的，因为有些发行版的 Docker 版本太低了，当然也可以手动升级 Docker，然后安装 Docker Compose。目前最新的版本 Docker Compose 1.9 要求 Docker 的版本最低为 1.10。

12.1.2 使用 Python pip 安装

先安装 Python（不同发行版安装方式不一样，以 Debian 系为例）。

```
$ sudo apt-get install python
```

使用 pip 安装需要确保已经安装了 pip 工具，安装 pip。

```
$ curl https://bootstrap.pypa.io/get-pip.py | python
```

如果已经安装了这些工具，那么可以直接安装 Docker Compose 了。

```
$ sudo pip install -U docker-compose
```

使用 docker-compose -v 查看是否安装成功：

```
$ docker-compose -v
docker-compose version 1.8.0, build f3628c7
```

12.2 Compose 命令基础

学习 Docker Compose 当然不能忽视最基本的启动参数。Docker Compose 命令是 docker-compose，执行 --help 就可以查看帮助信息。

Docker Compose 默认解析当前目录的 docker-compose.yml 文件，Docker Compose 的命令有些类似 Docker client 的子命令，如下：

```
$ docker-compose
Define and run multi-container applications with Docker.

Usage:
  docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help

Options:
  -f, --file FILE                Specify an alternate compose file (default:
docker-compose.yml)
  -p, --project-name NAME        Specify an alternate project name (default:
directory name)
  --verbose                      Show more output
# 输出更多信息可以使用 --verbose 参数
  -v, --version                 Print version and exit
# 查看版本信息
  -H, --host HOST               Daemon socket to connect to
# 设置 Daemon 的 socket 地址
  --tls                          Use TLS; implied by --tlsverify
  --tlscacert CA_PATH           Trust certs signed only by this CA
  --tlscert CLIENT_CERT_PATH    Path to TLS certificate file
  --tlskey TLS_KEY_PATH         Path to TLS key file
  --tlsverify                   Use TLS and verify the remote
  --skip-hostname-check          Don't check the daemon's hostname against the
                                name specified
                                in the client certificate (for example if your
                                docker host
                                is an IP address)

Commands:
  .....
  help                          Get help on a command
  .....
  version                      Show the Docker-Compose version information
```

Docker Compose 的子命令在稍后介绍，下面先看 Commands 上面的两个常用参数。

12.2.1 指定配置文件

`-f` 参数是用来指定 Docker Compose 的配置文件。这个参数可以使用多次，例如：

```
$ docker-compose -f docker-compose.yml -f docker-compose.admin.yml run backup_db
```

如果两份配置文件有同名的服务，Docker Compose 只会解析执行后面的配置文件。例如在 `docker-compose.yml` 中有一个服务叫做 `webapp`：

```
# docker-compose.yml
webapp:
  image: examples/web
  ports:
    - "8000:8000"
  volumes:
    - "/data"
```

在 `docker-compose.admin.yml` 也有一个叫做 `webapp` 的服务：

```
# docker-compose.admin.yml
webapp:
  build: .
  environment:
    - DEBUG=1
```

Docker Compose 会执行后面的 `webapp` 配置。`-f` 选项是可选的，如果不使用该选项，默认会解析当前目录下的 `docker-compose.yml` 文件。

12.2.2 指定项目名称

Docker Compose 启动容器时会默认把当前的目录名称设置为容器名称前缀，例如在 Web 文件夹下启动容器，配置文件中有两个服务分别是 `app` 和 `db`，启动的容器名称默认是 `web_db_1` 和 `web_app_1`，如果想要指定容器项目名称（就是 `web` 这个前缀），可以使用 `-p` 参数：

```
$ docker-compose -p myapp up
myapp_db_1
myapp_app_1
```

当然如果想完全指定容器名称，可以在配置文件中设置，后面会介绍到该内容。

除了上面两个参数，还有其他参数，但是一般用不到，因此不做介绍。

12.2.3 Compose 环境变量

在 Docker Compose 中有一个环境配置文件 `.env`，这是一个隐藏文件，文件中可以设定一些 Docker Compose 的环境变量：

```
COMPOSE_API_VERSION
COMPOSE_FILE
COMPOSE_HTTP_TIMEOUT
COMPOSE_PROJECT_NAME
DOCKER_CERT_PATH
```

```
DOCKER_HOST
DOCKER_TLS_VERIFY
```

- **COMPOSE_PROJECT_NAME**: 该变量用来定义使用 Compose 启动容器时的名称, 作用与 **-p** 相同。
- **COMPOSE_FILE**: 指定默认的配置文件的名称, 默认是 `docker-compose.yml`, 作用类似 **-f** 选项。
- **DOCKER_HOST**: 指定 Docker client 连接 Docker daemon 的地址, 默认是 `unix:///var/run/docker.sock`。

如果是远程操作, 可以使用上面的 `--tls*` 选项, 确保指令传输过程加密, 因为有时候启动命令中会有明文密码。

12.2.4 构建服务镜像的 build 命令

Docker Compose 提供了类似 Docker client 的构建命令, 与 `docker run` 不同, `docker-compose.yml` 不只是一份启动配置, 有时还包括构建定义, 例如:

```
mysql:
  build: ./db/
  restart: always
  volumes:
  - /data/database:/var/lib/mysql
ui:
  build:
    context: ../
    dockerfile: myapp/ui/Dockerfile
  restart: always
  ports:
  - "9090:80"
```

上面的 `docker-compose.yml` 中在执行 `docker-compose build` 命令时会自动构建 MySQL 与 `ui` 两个镜像, 默认构建的镜像名称是 `myapp_mysql` 和 `myapp_ui`。上面在设置 Dockerfile 路径的同时还可以指定 Dockerfile 的上下文路径, 这是 Docker client 做不到的。

在 `docker-compose.yml` 里面定义构建镜像时要注意把 Dockerfile 写好, 因为 Docker Compose 实际上是通过 `docker-compose.yml` 读取信息解析后发给 Docker client 执行的。在 `docker-compose.yml` 中可以使用相对路径。`docker-compose.yml` 的语法会在后面介绍。

在 `docker-compose.yml` 中通常包含了多个容器构建、启动配置, 默认情况下使用 `docker-compose build` 是构建 `docker-compose.yml` 里面的所有镜像。但是有时候只是想构建其中一个容器的镜像, 这时候可以指定构建容器名称, 例如:

```
$ docker-compose build ui
```

这个命令适用于重构部分镜像时使用, 避免重构全部镜像。

使用 `--help` 查看帮助信息会发现, `build` 命令还有 3 个选项:

```
$ docker-compose build --help
Build or rebuild services.
```

```
Services are built once and then tagged as 'project_service',
e.g. 'composetest_db'. If you change a service's 'Dockerfile' or the
contents of its build directory, you can run 'docker-compose build' to rebuild
it.
```

Usage: build [options] [SERVICE...]

Options:

--force-rm Always remove intermediate containers.
 --no-cache Do not use cache when building the image.
 --pull Always attempt to pull a newer version of the image.

--force-rm 和 --no-cache 都是属于在构建过程中自动清理缓存的选项，我们知道，构建过程实际是后台运行一个容器在执行 Dockerfile 的指令，这其中会产生中间容器，使用 --force-rm 选项会在构建结束时删除这些容器。如果构建失败时，Docker 会保留上一个临时容器在本地，该容器保存了直至失败的指令前面的构建内容，也就是我们说的构建缓存。使用 --no-cache 会在构建过程中自动删除构建缓存，上面这两个选项都不推荐使用。

此外还有一个选项就是 --pull 选项，默认情况下 Docker Compose 会在启动容器的时候查看本地是否有该镜像，如果有就直接使用本地已存在的镜像，使用 --pull 之后即使本地有该镜像，也会执行该 pull 命令拉取镜像，这样可以确保每次启动的容器都是基于最新的镜像启动。

12.2.5 生成 DAB 包的 bundle 命令

从 docker-compose.yml 文件中生成一个分布式应用程序包（DAB），这个概念涉及分布式应用，本章不会过多介绍，简单来说就是会生成一个 .dab 的文件，然后可以使用 docker deploy 来部署，这个 deploy 功能还处于 experimental 状态，没有发布到正式版本中，因此本书不做介绍。有关介绍可以看 <https://blog.docker.com/2016/06/docker-app-bundle/>。

12.2.6 检查配置语法的 config 命令

config 命令用来检查 docker-compsoe.yml 文件是否有语法问题，如果有会返回错误原因。

```
$ docker-compose config --help
Validate and view the compose file.
```

Usage: config [options]

Options:

-q, --quiet Only validate the configuration, don't print anything.
 --services Print the service names, one per line.

例如，下面一个简单的应用有两个服务，直接使用 config 命令会输出 docker-compose.yml 的内容：

```
$ nginx docker-compose config
networks:
  default:
    external:
      name: nginx_default
services:
  app:
    container_name: bbs
    image: abiosoft/caddy:php
```

```

    ports:
    - 2015:2015
    restart: always
  nginx:
    container_name: nginx
    image: nginx:alpine
    ports:
    - 80:80
    - 443:443
    restart: always
version: '2.0'
volumes: {}

```

使用-q 选项检查时，不会输出任何信息，除非有语法问题。

```
$ nginx docker-compose config -q
```

使用--service 选项时会输出服务名称。

```
$ nginx docker-compose config --service
acg
bbs
nginx

```

12.2.7 创建服务容器的 create 命令

create 命令与 docker create 类似，使用 docker-compose create 命令会创建所有服务需要的容器，但是不会运行容器。

```
$ docker-compose create --help
Creates containers for a service.
```

```
Usage: create [options] [SERVICE...]
```

Options:

```

--force-recreate      Recreate containers even if their configuration
and                  image haven't changed. Incompatible with --no-recreate.
// 即使容器配置和镜像没有变动也重新创建容器。与 --no-recreate 不兼容
--no-recreate         If containers already exist, don't recreate them.
                      Incompatible with --force-recreate.
// 如果容器已经存在，就不再重新创建它们。与 --force-recreate 不兼容
--no-build            Don't build an image, even if it's missing.
// 即使镜像不存在也不构建镜像
--build               Build images before creating containers.
// 在创建容器之前构建图像

```

create 命令相对比较简单，只是创建容器，不会占用存储之外的硬件资源。

12.2.8 清理项目的 down 命令

down 命令与后面的 up 命令相对，down 命令可以停止容器并删除包括容器、网络、数据卷等内容。也就是只要是 up 命令创建的东西，使用 down 命令都可以删除。此外，如果网络、数据卷等资源正在被其他服务使用，down 命令会跳过这些组件。例如：

```
$ docker-compose down
Stopping myapp_app_1 ... done
```

```
Stopping myapp_db_1 ... done
Removing myapp_app_1 ... done
Removing myapp_db_1 ... done
Network nginx_default is external, skipping
```

与 `docker rm` 类似, `docker-compose down` 也可以通过 `-v` 和 `--rmi` 来指定删除的内容。

```
$ docker-compose down --help
Stops containers and removes containers, networks, volumes, and images
created by 'up'.
```

By default, the only things removed are:

- Containers for services defined in the Compose file
- Networks defined in the 'networks' section of the Compose file
- The default network, if one is used

Networks and volumes defined as 'external' are never removed.

Usage: down [options]

Options:

- `--rmi type` Remove images. Type must be one of:
 'all': Remove all images used by any service.
 'local': Remove only images that don't have a custom tag
 set by the 'image' field.
- `-v, --volumes` Remove named volumes declared in the 'volumes' section
 of the Compose file and anonymous volumes
 attached to containers.
- `--remove-orphans` Remove containers for services not defined in the
 Compose file

默认情况下, `down` 命令只会删除定义的服务运行的容器以及网络。通过指定 `-v` 参数会删除数据卷, 指定 `--rmi` 可以删除与服务相关的镜像。此外, 使用 `--remove-orphans` 还可以删除与服务相关, 但是没有在配置文件重定义的容器。

12.2.9 查看事件的 events 命令

`events` 命令实际上就是对 `docker events` 的整合, 通过该命令可以看到与配置文件定义的服务的相关事件。

```
$ docker-compose events --help
Receive real time events from containers.
```

Usage: events [options] [SERVICE...]

Options:

- `--json` Output events as a stream of json objects

关于事件的定义在第 4 章中已经讲过, 这里不再赘述。可以通过该命令查看项目中运行容器的实时事件流。使用 `--json` 选项可以格式化输出, 更容易阅读。

```
{
  "service": "web",
  "event": "create",
  "container": "213cf75fc39a",
  "image": "alpine:edge",
  "time": "2015-11-20T18:01:03.615550",
}
```

12.2.10 进入服务的 exec 命令

`exec` 命令与 `docker exec` 命令类似，可以进入容器执行命令，不同的是 `docker-compose exec` 后面是服务名称而不是容器名称。

```
$ docker-compose exec --help
Execute a command in a running container

Usage: exec [options] SERVICE COMMAND [ARGS...]

Options:
  -d                Detached mode: Run command in the background.
  --privileged      Give extended privileges to the process.
  --user USER       Run the command as this user.
  -T                Disable pseudo-tty allocation. By default
'docker-compose exec'
                    allocates a TTY.
  --index=index     index of the container if there are multiple
                    instances of a service [default: 1]
```

使用 `-d` 参数可以在后台执行命令；使用 `--privileged` 选项可以开启特权模式，获得宿主机的 `root` 权限；`--user` 可以切换进入容器时的用户身份；`-T` 参数可以禁用 `pseudo-tty` 分配，默认情况下会分配 `tty`，类似于自动加上 `-t` 参数的 `docker exec`，如果不需要可以使用 `-T` 禁用；`--index` 可以指定容器索引值，在一些服务中会启动多个相同容器实例来确保访问正常（`myapp_1`、`myapp_2`……），`index` 的值有助于负载与高可用，指定之后可以进入相应索引的容器里面。

12.2.11 杀死服务容器的 kill 命令

使用 `kill` 命令，默认会杀死项目下所有服务的容器。如果指定服务名称，则可以杀死指定服务下的容器，不可以杀死指定容器名称。

```
$ docker-compose kill --help
Force stop service containers.

Usage: kill [options] [SERVICE...]

Options:
  -s SIGNAL          SIGNAL to send to the container.
                    Default signal is SIGKILL.
```

使用 `-s` 参数可以改变发送的信号为 `SIGNAL`，默认为 `SIGKILL`。

12.2.12 查看服务容器日志的 logs 命令

`logs` 命令用于查看项目日志，默认这些日志包含了全部容器的日志，输出时会用不同的颜色标示，指定服务名称可以查看指定服务的日志。

```
$ docker-compose logs --help
View output from containers.
```



```
Usage: logs [options] [SERVICE...]
```

Options:

```
--no-color      Produce monochrome output.
-f, --follow     Follow log output.
-t, --timestamps Show timestamps.
--tail="all"     Number of lines to show from the end of the logs
                  for each container.
```

使用 `--no-color` 可以取消颜色标示，一般建议保留；`-f` 可以保持输出不中断，也就是一直显示下去，除非使用 `Ctrl + C` 终止；`-t` 可以显示一个时间戳在每行日志前面，这样方便确定日志事件发生的时间；`--tail` 可以设定显示最后几行，参数值的数字表示显示日志最后的几行，默认显示全部。

12.2.13 暂停服务容器的 `pause` 命令

暂停项目服务可以使用 `pause` 命令，默认会停止全部的服务容器的进程，类似使用 `docker pause` 时的效果，如果需要停止指定的服务，可以在后面指明服务名称。

```
$ docker-compose pause --help
Pause services.
```

```
Usage: pause [SERVICE...]
```

使用了该命令的服务就像一个加锁的容器，即使使用 `kill` 也不能杀死，需要用 `unpause` 恢复进程才可以继续对容器操作。

12.2.14 查看服务容器端口状态的 `port` 命令

在 Docker Compose 中 `port` 命令不如 Docker client 中的 `port` 命令那么灵活，在 Docker Compose 中使用 `port` 命令，不仅需要指定服务名称，还需要指定服务容器暴露的端口，才可以查看该端口在宿主机中的映射。

```
$ docker-compose port --help
Print the public port for a port binding.
```

```
Usage: port [options] SERVICE PRIVATE_PORT
```

Options:

```
--protocol=proto tcp or udp [default: tcp]
--index=index      index of the container if there are multiple
                    instances of a service [default: 1]
```

`--protocol` 参数可以指定显示 `tcp` 或者是 `udp` 端口；`--index` 的用途是指定容器索引值，在一些服务中会启动多个相同容器实例来确保访问正常，指定之后可以显示该容器端口映射情况。

12.2.15 查看项目容器信息 `ps` 命令

`ps` 命令用途与 `docker ps` 类似，使用 `docker-compose ps` 可以查看正在运行的服务容器。

```
$ docker-compose ps --help
```

List containers.

Usage: ps [options] [SERVICE...]

Options:

-q Only display IDs

该命令只有一个参数，-q 输出容器 ID，在一些脚本中非常有用。

查看项目的服务容器：

```
$ docker-compose ps
```

Name	Command	State	Ports
app1	/usr/bin/caddy --conf /etc ...	Up	0.0.0.0:2333->2015/tcp, 443/tcp, 80/tcp
app2	/usr/bin/caddy --conf /etc ...	Up	0.0.0.0:2015->2015/tcp, 443/tcp, 80/tcp
nginx	nginx -g daemon off;	Up	0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp

查看指定服务容器：

```
$ nginx docker-compose ps nginx
```

Name	Command	State	Ports
nginx	nginx -g daemon off;	Up	0.0.0.0:443->443/tcp, 0.0.0.0:80->80/tcp

只显示容器 ID：

```
$ nginx docker-compose ps -q
```

```
efbafad9b1aa6f38b985a23b8be2a86d4aed552f814131417ff8d380304e3917
1c52baf20f208b22f4a9f2580c5d5e1ee6116340a1d37256a76d8eec18415d0d
b100008b1841df269fbc04461aecbf2fb0b8c58cd4fec074c11057a03a427342
```

12.2.16 拉取项目镜像的 pull 命令

使用 docker-compose pull 可以拉取多个镜像，因为在一份 docker-compose.yml 文件中通常有多个服务，每个服务要有一个镜像作为镜像基础。

在 Docker Compose 所有子命令中都可以使用 -f 这些参数，因此在 pull 操作时也可以指定多个配置文件，如果服务名相同，后来者会覆盖前者，pull 操作只会拉取后出现的服务所需要的镜像。

```
$ docker-compose pull --help
```

Pulls images for services.

Usage: pull [options] [SERVICE...]

Options:

--ignore-pull-failures Pull what it can and ignores images with pull failures.

参数 --ignore-pull-failures 可以无视拉取失败的提示，继续执行下去，如果没有这个参数则会在拉取失败时自动终止后面的镜像拉取操作。

12.2.17 推送项目镜像的 push 命令

在一些项目中，镜像并不是基于现成的 Docker 镜像运行的，而是在第一次启动的时候自动创建的，因此项目中有新构建的镜像时，可以使用 `push` 命令推送项目的服务镜像到仓库。

```
$ docker-compose push --help
Pushes images for services.
```

```
Usage: push [options] [SERVICE...]
```

Options:

```
--ignore-push-failures Push what it can and ignores images with push
failures.
```

参数 `--ignore-push-failures` 可以无视推送失败的提示，继续执行下去，如果没有这个参数则会在遇到推送失败时自动终止后面的镜像拉取操作。

12.2.18 重启服务容器的 restart 命令

在 Docker Compose 中可以像 Docker client 一样操作容器，所以当然也包括重启服务容器，`restart` 默认会重启项目下的全部服务容器。

如果用户指定服务名称，可以重启指定的服务容器。

```
$ docker-compose restart --help
Restart running containers.
```

```
Usage: restart [options] [SERVICE...]
```

Options:

```
-t, --timeout TIMEOUT Specify a shutdown timeout in seconds.
                        (default: 10)
```

`-t` 参数与之前介绍的一样，在设定秒数之内没有响应就会发送 `SIGKILL` 信号停止容器。

12.2.19 删除项目容器的 rm 命令

使用 `rm` 命令当然是可以删除服务容器了，Docker Compose 的 `rm` 实际上就是对配置文件解析后向 Docker client 发送 `rm` 的 API 请求，因此 Client 的参数在 Compose 里也大都适用，例如 `-v` 是删除服务容器的数据卷。

```
$ docker-compose rm --help
Removes stopped service containers.
```

```
By default, anonymous volumes attached to containers will not be removed.
You can override this with '-v'. To list all volumes, use 'docker volume
ls'.
```

```
Any data which is not in a volume will be lost.
```

```
Usage: rm [options] [SERVICE...]
```

Options:

```
-f, --force    Don't ask to confirm removal
-v            Remove any anonymous volumes attached to containers
-a, --all      Obsolete. Also remove one-off containers created by
               docker-compose run
```

-f 是强制删除服务容器，与 Docker client 不同的是，Compose 不允许强制删除正在运行的容器，因此必须是停止或者杀死容器之后才能执行删除容器的操作。使用 **docker-compose rm** 操作时默认会提示是否真的删除容器：

```
$ docker-compose rm
No stopped containers
$ docker-compose stop
Stopping web_app_1 ... done
Stopping web_db_1 ... done
$ docker-compose rm
Going to remove web_app_1, web_db_1
Are you sure? [yN]
```

而使用 **-f** 参数之后会直接删除不会询问。

-a 参数是一个过时的参数，未来会删除这个选项，作用是删除全部容器，就是现在默认的 **rm** 操作。

12.2.20 执行一次性命令的 run 命令

与 Docker client 不同的是，Compose 并没有给 **run** 命令太多的可选参数。Compose 的 **run** 命令与 Docker client 的 **run** 命令不一样。使用 **docker-compose run** 命令只能对一个服务的容器运行一次一次性的命令。例如，启动一个容器的 **bash** 命令：

```
$ docker-compose run app bash
```

使用 **run** 指令运行容器时，创建的容器不属于项目中的服务，而是作为一个独立的容器，例如下面项目有两个服务，一个 **app** 和一个 **db**，**app** 依赖 **db** 服务，现在需要使用 **app** 的配置来临时执行一条一次性的命令，这时候就需要用 **run** 命令来运行一个 **app** 容器：

```
$ docker-compose run -d app ./script.sh
web_app_run_1
```

可以看到使用 **run** 命令执行时创建的容器不属于项目服务的一部分，容器名字表明了这是一个使用 **run** 命令启动的一次性容器。执行删除时也不像上面的 **rm** 命令一样必须要停止容器才可以删除，而是直接删除。

```
$ docker-compose rm -f
Going to remove web_app_run_1
Removing web_app_run_1 ... Done
```

如果同时启动多个一次性容器，会生成多个 **run** 标志的容器。

```
$ docker-compose run -d app
web_app_run_1
$ docker-compose run -d app
web_app_run_2
```

但是使用 **ps** 命令查看服务时是看不到这些容器信息的。

```
$ docker-compose ps
```

Name	Command	State	Ports
------	---------	-------	-------

```
-----
wekan_db_1  /entrypoint.sh mongod Up      27017/tcp
```

使用 `rm -f` 命令时会直接删除所有一次性容器。

```
$ docker-compose rm -f
Going to remove wekan_app_run_2, wekan_app_run_1
Removing wekan_app_run_2 ... done
Removing wekan_app_run_1 ... done
```

这个命令会使用配置文件里面定义的配置来启动容器，这意味着启动容器具有相同的数据卷、链接映射和配置，但是有以下两点不同。

第一点，`run` 会覆盖配置文件中的运行指令，例如容器默认 `CMD` 是 `bash`，使用 `docker-compose run app python` 之后 `python` 会覆盖 `bash` 命令。

第二点，`run` 命令不会解析执行配置文件中的端口映射定义，这可以有效防止端口占用等问题，如果需要在 `run` 命令中执行端口映射，可加上 `--service-ports` 参数，或者手动指定端口映射，和 `Docker client` 一样使用 `-p` 参数。

```
$ docker-compose run --publish 8080:80 -p 2022:22 -p 127.0.0.1:2021:21 .....
```

```
$ docker-compose run --help
Run a one-off command on a service.
```

For example:

```
$ docker-compose run web python manage.py shell
```

By default, linked services will be started, unless they are already running. If you do not want to start linked services, use `'docker-compose run --no-deps SERVICE COMMAND [ARGS...]'`.

Usage: `run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]`

Options:

```
-d                      Detached mode: Run container in the background, print
                        new container name.
```

// 在后台运行命令可以使用 `-d` 参数

```
--name NAME            Assign a name to the container
```

// 指定容器名称

```
--entrypoint CMD       Override the entrypoint of the image.
```

// 覆盖 `Dockerfile` 中的 `ENTRYPOINT` 指令

```
-e KEY=VAL             Set an environment variable (can be used multiple
times)
```

// 设置环境变量

```
-u, --user="..."      Run as specified username or uid
```

// 设置执行命令的用户身份

```
--no-deps              Don't start linked services.
```

// 取消容器关联，详见下面解释

```
--rm                  Remove container after run. Ignored in detached mode.
```

// 执行完命令之后自动删除容器

```

-p, --publish=[]      Publish a container's port(s) to the host
// 手动指定端口映射
--service-ports        Run command with the service's ports enabled and
                        mapped
                        to the host.
// 解析执行配置文件中的端口映射定义

-T                    Disable pseudo-tty allocation. By default 'docker-
                        compose run'
                        allocates a TTY.
// 不显示 TTY, 默认申请 TTY

-w, --workdir=""      Working directory inside the container
// 设置执行命令的目录, 覆盖 Dockerfile 的 WORKDIR 指令

```

上面提到一个 `--no-deps` 参数, 这是一个取消容器关联的参数。例如有一个项目, 项目内有两个服务, 其中一个应用容器 (app), 另一个是数据库 (db), 容器依赖数据库容器, 如果使用 `docker-compose run app bash` 来运行一个一次性指令, 会默认启动数据库容器, 如果不需要这种关联就需要添加 `--no-deps` 参数了。

到底什么时候需要使用 `run` 命令呢? 一个简单的例子就是备份数据库, 我们知道, 如果服务写到配置文件中就会解析执行, 因此备份数据库的配置文件通常要独立开, 这个时候就有两个配置文件了, 如下面的例子。

在 `docker-compose.yml` 中定义 Web 和 db 两个服务:

```

web:
  image: example/my_web_app:latest
  links:
    - db

db:
  image: postgres:latest

```

然后在 `docker-compose.backup.yml` 中定义一个备份服务, 内容如下:

```

dbbackup:
  build: database_backup/
  links:
    - db

```

然后使用 `run` 命令来运行这次备份任务, 这是一次一次性命令, 使用 `-f` 加入两个配置文件才可以完成备份任务, 这样就把备份配置与运行配置分开了。

```

$ docker-compose -f docker-compose.yml \
  -f docker-compose.backup.yml \
  run dbbackup ./db_backup.sh

```

12.2.21 设置服务容器数量的 scale 命令

`scale` 命令体现了 Compose 的人性化特点, 通常 Web 服务为了保证高可用和负载均衡, 会在后端启动多个服务, 确保应用不会因为其中一个后端服务崩溃而无法访问。

而 `scale` 就是一个可以设置服务容器启动个数的命令, 使用格式如下:

```

$ docker-compose scale <服务名称>=<启动个数>

$ docker-compose scale --help

```


Set number of containers to run for a service.

Numbers are specified in the form 'service=num' as arguments.
For example:

```
$ docker-compose scale web=2 worker=3
```

Usage: scale [options] [SERVICE=NUM...]

Options:

-t, --timeout TIMEOUT Specify a shutdown timeout in seconds.
(default: 10)

-t 设置超时，超时就向容器进程发送 shutdown 信号关闭容器。

下面以一个简单的例子来说明 scale 命令的应用，如图 12.1 所示。

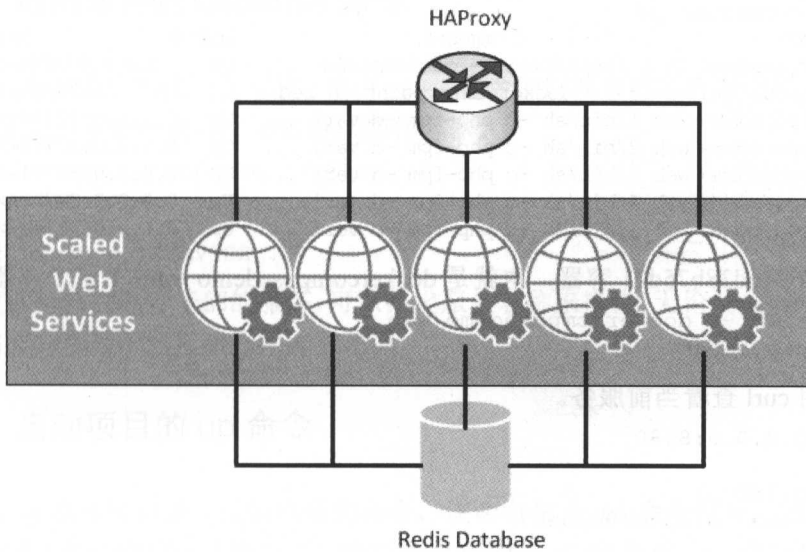


图 12.1 scale 模型

```
$ git clone https://github.com/vegasbrianc/docker-compose-demo.git
```

复制演示项目并启动。

```
$ cd docker-compose-demo
```

```
$ docker-compose up -d
```

```
Starting dockercomposedemo_web_1
```

```
Starting dockercomposedemo_redis_1
```

```
Starting dockercomposedemo_lb_1
```

查看项目状态。

```
$ docker-compose ps
```

Name	Command	State	Ports
dockercomposedemo_lb_1	/sbin/tini -- dockercloud- ...	Up	0.0.0.0:8080->80/tcp
dockercomposedemo_redis_1	docker-entrypoint.sh redis ...	Up	6379/tcp
dockercomposedemo_web_1	/bin/sh -c php-fpm -d vari ...	Up	80/tcp, 0.0.0.0:32770->8080/tcp

可以看到项目运行正常，通过 curl 获得的信息显示返回的容器名称是 716d38b75d21。

```
$ curl 0.0.0.0:8080
.....
Hello world!
My hostname is 716d38b75d21
.....
```

现在启动 5 个这样的 Web 服务。

```
$ docker-compose scale web=5
Creating and starting dockercomposedemo_web_2 ... done
Creating and starting dockercomposedemo_web_3 ... done
Creating and starting dockercomposedemo_web_4 ... done
Creating and starting dockercomposedemo_web_5 ... done
```

因为本来有一个 Web 服务了，所以上面启动了 4 个，查看一下状态，发现有 5 个 Web 服务了。

```
$ docker-compose ps
      Name                                Command                                State      Ports
dockercomposedemo_lb_1 /sbin/tini -- dockercloud- ... Up        0.0.0.0:8080->80/tcp
dockercomposedemo_redis_1 docker-entrypoint.sh redis ... Up        6379/tcp
dockercomposedemo_web_1/bin/sh -c php-fpm -d vari ... Up        0.0.0.0:32770->8080/tcp
dockercomposedemo_web_2/bin/sh -c php-fpm -d vari ... Up        0.0.0.0:32775->8080/tcp
dockercomposedemo_web_3/bin/sh -c php-fpm -d vari ... Up        0.0.0.0:32774->8080/tcp
dockercomposedemo_web_4/bin/sh -c php-fpm -d vari ... Up        0.0.0.0:32773->8080/tcp
dockercomposedemo_web_5/bin/sh -c php-fpm -d vari ... Up        0.0.0.0:32772->8080/tcp
```

现在删除 716d38b75d21 容器，也就是 dockercomposedemo_web_1 这个容器。

```
$ docker rm -f dockercomposedemo_web_1
dockercomposedemo_web_1
```

然后使用 curl 查看当前服务。

```
$ curl 0.0.0.0:8080
.....
Hello world!
My hostname is d57bb9babeb7
.....
```

可以看到服务并没有崩溃，而是其他容器顶替上来，返回同样的 Hello World 信息，但是实际上容器已经变了。

12.2.22 启动服务容器的 start 命令

start 是启动服务的命令，可以启动非运行的容器，默认会启动所有服务容器，指定服务名称可以启动指定服务。

```
$ docker-compose start --help
Start existing containers.
```

```
Usage: start [SERVICE...]
```

该命令使用的前提是容器已经存在。

12.2.23 停止服务容器的 stop 命令

stop 显然是停止容器的命令，该命令只会停止容器，不会删除容器。默认会停止全部

服务容器。可以指定服务名称来停止相应的服务。

```
$ docker-compose stop --help
Stop running containers without removing them.

They can be started again with 'docker-compose start'.

Usage: stop [options] [SERVICE...]

Options:
  -t, --timeout TIMEOUT    Specify a shutdown timeout in seconds.
                           (default: 10)
```

-t 参数与前面介绍的一样，设置响应等待时间，超时直接“杀死”容器。

12.2.24 取消暂停的 unpause 命令

unpause 命令在前面已经介绍过，使用 pause 命令的时候会锁定容器进程，这时候需要使用 unpause 命令来取消暂停才可以继续操作容器。

```
$ docker-compose unpause --help
Unpause services.

Usage: unpause [SERVICE...]
```

同样默认是针对项目全部的服务，可以指定服务名称来操作。无论是 pause 还是 unpause 命令，都需要容器处于运行状态才能操作。

12.2.25 启动项目的 up 命令

最后的 up 命令可以说是压轴出场的命令，该命令与 down 命令相对，使用 up 命令的时候会从配置文件读取解析各项定义，然后发给 Docker client 执行，up 命令可以创建包括服务容器、数据卷、网络等一系列组件，这也是经常使用的 Compose 命令，可以说万事从 up 开始。

```
$ docker-compose up --help
Builds, (re)creates, starts, and attaches to containers for a service.

Unless they are already running, this command also starts any linked services.
.....
If you want to force Compose to stop and recreate all containers, use the
'--force-recreate' flag.
Usage: up [options] [SERVICE...]

Options:
  // 在后台运行服务
  -d                               Detached mode: Run containers in the background,
                                   print new container names.
                                   Incompatible with --abort-on-container-exit.

  // 输出时不显示颜色，建议使用颜色区分
  --no-color                       Produce monochrome output.

  // 启动时不建立容器连接
  --no-deps                       Don't start linked services.
```

// 即使配置文件和镜像没有变动,也会重新创建容器并启动。与 `--no-recreate` 参数冲突。
默认情况下,当配置或者镜像发生变动时,会重新创建容器

```
--force-recreate      Recreate containers even if their configuration
                        and image haven't changed.
                        Incompatible with --no-recreate.
```

```
// 如果容器存在,则不会重新创建容器,与 --force-recreate 参数冲突
--no-recreate          If containers already exist, don't recreate them.
                        Incompatible with --force-recreate.
```

```
// 使用这个参数时,即便镜像不存在也不会从 Dockerfile 中构建
--no-build              Don't build an image, even if it's missing.
```

```
// 启动容器之前先构建镜像
--build                 Build images before starting containers.
```

```
// 如果项目中有一个容器退出了,其他容器也会被停止,这个参数与 -d 冲突
--abort-on-container-exit Stops all containers if any container was stopped.
                        Incompatible with -d.
```

```
// 设定超时
-t, --timeout TIMEOUT  Use this timeout in seconds for container shutdown
                        when attached or when containers are already
                        running. (default: 10)
```

```
// 删除与服务相关但是未在配置文件中定义的容器
--remove-orphans        Remove containers for services not
                        defined in the Compose file
```

Compose 的 `up` 命令包括了构建、创建（重新创建）、启动和连接服务容器。

直接使用 `docker-compose up` 命令可以聚合全部的容器信息,每个容器输出内容会用不同的颜色区分,当使用 `Ctrl+C` 命令停止时,会停止所有容器。

如果想让项目在后台运行,就需要添加 `-d` 参数,这样服务就会在后台运行,通过 `docker-compose ps` 可以查看容器运行状态, `logs` 可以看到所有容器的日志。

12.3 Compose 配置文件

Docker Compose 是使用 YAML 文件来定义多个容器关系的,因此掌握 `docker-compose.yml` 文件的写法才能更好地书写配置文件,方便管理多容器应用。

Docker Compose 实际上是把 YAML 文件解析成原生的 Docker 命令然后执行的,它通过定义解析容器依赖关系来按顺序启动容器。

12.3.1 配置文件基础

Compose 配置文件是一个 YAML 格式的文件,它定义了包括服务（容器）、网络、数据卷在内的一系列项目组件。默认的 Compose 配置文件路径是 `./docker-compose.yml`。

使用配置文件定义的服务在启动时就像使用 Docker client 的 `docker run` 一样,同样的配置文件重定义的网络、数据卷也相当于 Docker client 中使用 `docker network create` 和

`docker volume create` 一样。实际上，Compose 并不会真正地操作容器，它管理容器的办法就是解析配置文件的定义，然后发送给 Docker client。

Compose 配置文件中定义的每个服务都必须通过 `image` 标签指定镜像或 `build` 标签来执行构建（上下文中存在 `Dockerfile`）。其实配置文件的写法与 Docker client 中的命令有异曲同工之妙。

就像 `docker run` 命令中一样，使用 Compose 命令时，`Dockerfile` 中的指令依然有效，不必在 `docker-compose.yml` 文件中重新设定。

例如，在 `Dockerfile` 中定义的变量可以在 `docker-compose.yml` 文件中使用，就像 shell 脚本的写法一样，形如 `${EXTERNAL_PORT}` 即可。

因为历史原因，Compose 配置文件有两个版本，本书不对 v1 进行讲解，默认全都是 v2 的配置文件写法。

12.3.2 基本配置

先来看前面 `scale` 命令中的 `demo` 项目的 `docker-compose.yml` 文件：

```
version: '2'
services:
  web:
    image: dockercloud/hello-world
    ports:
      - 8080
    networks:
      - front-tier
      - back-tier

  redis:
    image: redis
    links:
      - web
    networks:
      - back-tier

  lb:
    image: dockercloud/haproxy
    ports:
      - 80:80
    links:
      - web
    networks:
      - front-tier
      - back-tier
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

networks:
  front-tier:
    driver: bridge
  back-tier:
    driver: bridge
```

可以看到，一份标准配置文件应该包含 `version`、`services`、`networks` 这 3 大部分，其中最关键的就是 `services` 和 `networks` 两个部分，下面先来看 `services` 的书写规则。

1. 指定服务使用的镜像: image

在 `services` 标签下的第二级标签是 `Web`, 这个名字是用户自己自定义, 它就是服务名称。

```
services:
  web:
    image: hello-world
```

在 `services` 标签下的第二级标签是 `web`, 这个名字是用户自己定义, 它就是服务名称。

`image` 则是指定服务的镜像名称或镜像 ID。如果镜像在本地不存在, `Compose` 将会尝试拉取这个镜像。例如, 下面这些格式都是可以的:

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

2. 指定构建上下文: build

服务除了可以基于指定的镜像, 还可以基于一份 `Dockerfile`, 在使用 `up` 命令启动之时执行构建任务, 这个构建标签就是 `build`, 它可以指定 `Dockerfile` 所在文件夹的路径。`Compose` 将会利用它自动构建这个镜像, 然后使用这个镜像启动服务容器。

```
build: /path/to/build/dir
```

也可以是相对路径, 只要上下文确定就可以读取到 `Dockerfile`。

```
build: ./dir
```

设定上下文根目录, 然后以该目录为准指定 `Dockerfile`。

```
build:
  context: ../
  dockerfile: path/of/Dockerfile
```

注意 `build` 都是一个目录, 如果要指定 `Dockerfile` 文件, 需要在 `build` 标签的子级标签中使用 `dockerfile` 标签指定, 如上面的例子。

如果同时指定了 `image` 和 `build` 两个标签, 那么 `Compose` 会构建镜像并且把镜像命名为 `image` 后面的那个名字。

```
build: ./dir
image: webapp:tag
```

既然可以在 `docker-compose.yml` 中定义构建任务, 那么一定少不了 `arg` 这个标签, 还记得 `Dockerfile` 中的 `ARG` 指令吧, 它可以在构建过程中指定环境变量, 但是在构建成功后取消, 在 `docker-compose.yml` 文件中也支持这样的写法:

```
build:
  context: .
  args:
    buildno: 1
    password: secret
```

下面这种写法也是支持的, 一般来说这种写法更适合阅读。

```
build:
  context: .
  args:
```




```
- buildno=1
- password=secret
```

与 ENV 不同的是, ARG 是允许空值的。例如:

```
args:
  - buildno
  - password
```

这样构建过程可以给它们赋值。

 **注意:** YAML 的布尔值 (true, false, yes, no, on, off) 必须要使用引号引起来 (单引号、双引号均可), 否则会当成字符串解析。

3. 指定服务镜像启动命令: command

使用 command 可以覆盖容器启动后默认执行的命令。

```
command: bundle exec thin -p 3000
```

也可以写成类似 Dockerfile 中的格式:

```
command: [bundle, exec, thin, -p, 3000]
```

4. 指定运行服务的容器名称: container_name

前面说过 Compose 的容器名称格式是<项目名称>_<服务名称>_<序号>。虽然可以自定义项目名称、服务名称, 但是如果想完全控制容器的命名, 可以使用以下标签指定:

```
container_name: app
```

这样容器的名字就指定为 app 了。

5. 指定服务依赖关系: depends_on

在使用 Compose 时, 最大的好处就是减少使用烦琐的启动命令, 但是一般项目容器启动的顺序是有要求的, 如果直接从上到下启动容器, 必然会因为容器依赖问题而启动失败。

例如, 在没启动数据库容器的时候启动了应用容器, 这时候应用容器会因为找不到数据库而退出, 为了避免这种情况, 我们需要加入一个标签, 就是 depends_on, 该标签解决了容器的依赖、启动先后的问题。

例如, 下面容器会先启动 redis 和 db 两个服务, 最后才启动 Web 服务:

```
version: '2'
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

需要注意的是, 默认情况下使用 docker-compose up web 方式启动 Web 服务时, 也会启动 redis 和 db 两个服务, 因为在配置文件中定义了依赖关系。

6. 指定服务的DNS配置: dns

和--dns 参数一样用途, 格式如下:

```
dns: 8.8.8.8
```

也可以是一个列表:

```
dns:
  - 8.8.8.8
  - 9.9.9.9
```

此外 dns_search 的配置也类似:

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

7. 挂载临时目录: tmpfs

挂载临时目录到容器内部, 与 run 的参数效果一样:

```
tmpfs: /run
tmpfs:
  - /run
  - /tmp
```

8. 指定服务镜像的接入点: entrypoint

在 Dockerfile 中有一个指令叫做 ENTRYPOINT 指令, 用于指定接入点, 第 4 章中曾对比过与 CMD 的区别。

在 docker-compose.yml 中可以定义接入点, 覆盖 Dockerfile 中的定义:

```
entrypoint: /code/entrypoint.sh
```

格式和 Docker 类似, 不过还可以写成以下这样:

```
entrypoint:
  - php
  - -d
    - zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-20100525/
    xdebug.so
  - -d
    - memory_limit=-1
  - vendor/bin/phpunit
```

9. 设置compose变量: env_file

还记得前面提到的.env 文件吧, 这个文件可以设置 Compose 的变量。而在 docker-compose.yml 中可以定义一个专门存放变量的文件。

如果通过 docker-compose -f FILE 指定了配置文件, 则 env_file 中路径会使用配置文件路径。

如果有变量名称与 environment 指令冲突, 则以后者为准。格式如下:

```
env_file: .env
```

或者根据 docker-compose.yml 设置多个:

```
env_file:
- ./common.env
- ./apps/web.env
- /opt/secrets.env
```

需要注意的是，这里所说的环境变量是对宿主机的 Compose 而言的，如果在配置文件中有 build 操作，这些变量并不会进入构建过程中，如果要在构建中使用变量，还是首选前面刚讲的 arg 标签。

10. 设置环境变量：environment

environment 与上面的 env_file 标签完全不同，反而和 arg 有几分类似，这个标签的作用是设置镜像变量，它可以保存变量到镜像里面，也就是说启动的容器也会包含这些变量设置，这是与 arg 最大的不同。

一般情况下 arg 标签的变量仅用在构建过程中。而 environment 和 Dockerfile 中的 ENV 指令一样会把变量一直保存在镜像、容器中，类似 docker run -e 的效果。

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:
```

```
environment:
- RACK_ENV=development
- SHOW=true
- SESSION_SECRET
```

11. 指定端口暴露：expose

expose 标签与 Dockerfile 中的 EXPOSE 指令一样，用于指定暴露的端口，但是只作为一种参考，实际上 docker-compose.yml 的端口映射还是需要 ports 这样的标签。

```
expose:
- "3000"
- "8000"
```

12. 选择项目外的容器：external_links

在使用 Docker 过程中，会有许多单独使用 docker run 启动的容器，为了使 Compose 能够连接这些不在 docker-compose.yml 中定义的容器，需要一个特殊的标签，就是 external_links，它可以让 Compose 项目里面的容器连接到那些项目配置外部的容器（前提是外部容器中必须至少有一个容器是连接到与项目内的服务的同一个网络里）。

格式如下：

```
external_links:
- redis_1
- project_db_1:mysql
- project_db_1:postgresql
```

13. 扩展连接服务器的hosts列表：extra_hosts

extra_hosts 为添加主机名的标签，就是向/etc/hosts 文件中添加一些记录，与 Docker client 的 --add-host 类似：

```
extra_hosts:
  - "somehost:162.242.195.82"
  - "otherhost:50.31.209.229"
```

启动之后查看容器内部 hosts:

```
162.242.195.82  somehost
50.31.209.229  otherhost
```

14. 添加元数据: labels

向容器添加元数据, 和 Dockerfile 的 LABEL 指令意思一样, 格式如下:

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""
labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

15. 设置容器互联: links

还记得前面的 depends_on 吧, 那个标签解决的是启动顺序问题, 而 links 标签解决的是容器连接问题, 与 Docker client 的 --link 效果一样, 会连接到其他服务中的容器。

格式如下:

```
links:
  - db
  - db:database
  - redis
```

使用的别名将会自动在服务容器中的/etc/hosts 里创建。例如:

```
172.12.2.186  db
172.12.2.186  database
172.12.2.187  redis
```

相应的环境变量也将被创建。

16. 配置服务日志: logging

logging 标签用于配置日志服务。格式如下:

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

默认的 driver 是 json-file。只有 json-file 和 journald 可以通过 docker-compose logs 显示日志, 其他方式有其他日志查看方式, 但目前 Compose 不支持。对于可选值可以使用 options 指定。有关更多这方面的信息可以阅读官方文档 <https://docs.docker.com/engine/admin/logging/overview/>。

17. 指定进程空间: pid

```
pid: "host"
```

将 PID 模式设置为主机 PID 模式, 与主机系统共享进程命名空间。容器使用 pid 标签


将能够访问和操纵其他容器和宿主机的名称空间。

18. 设置服务容器的端口映射: ports

ports 为映射端口的标签。

使用 HOST:CONTAINER 格式或者只是指定容器的端口, 宿主机会随机映射端口。

```
ports:
  - "3000"
  - "8000:8000"
  - "49100:22"
  - "127.0.0.1:8001:8001"
```

 **注意:** 当使用 HOST:CONTAINER 格式映射端口时, 如果使用的容器端口小于 60, 可能会得到错误得结果, 因为 YAML 将会解析 xx:yy 这种数字格式为 60 进制, 所以建议采用字符串格式。

19. 设置容器安全选项: security_opt

security_opt 为每个容器覆盖默认的标签。简单来说就是管理全部服务的标签。比如设置全部服务的 user 标签值为 USER。

```
security_opt:
  - label:user:USER
  - label:role:ROLE
```

20. 设置容器停止信息: stop_signal

设置另一个信号来停止容器。在默认情况下使用的是 SIGTERM 停止容器。设置另一个信号可以使用 stop_signal 标签。

```
stop_signal: SIGUSR1
```

21. 设置容器数据卷: volumes

挂载一个目录或者一个已存在的数据卷容器, 可以直接使用 [HOST:CONTAINER] 这样的格式, 或者使用 [HOST:CONTAINER:ro] 这样的格式, 后者对于容器来说, 数据卷是只读的, 这样可以有效保护宿主机的文件系统。

Compose 的数据卷指定路径可以是相对路径, 使用 . 或者 .. 来指定相对目录。

数据卷的格式可以是下面多种形式:

```
volumes:
  // 只是指定一个路径, Docker 会自动在创建一个数据卷 (这个路径是容器内部的)
  - /var/lib/mysql

  // 使用绝对路径挂载数据卷
  - /opt/data:/var/lib/mysql

  // 以 Compose 配置文件为中心的相对路径作为数据卷挂载到容器
  - ./cache:/tmp/cache

  // 使用用户的相对路径 (~ 表示的目录是 /home/<用户目录> 或者 /root/)
  - ~/configs:/etc/configs:ro
```

```
// 已经存在的命名的数据卷
- datavolume:/var/lib/mysql
```

如果不使用宿主机的路径，则可以指定一个 `volume_driver`。

```
volume_driver: mydriver
```

22. 挂载数据卷容器: `volumes_from`

从其他容器或者服务挂载数据卷，可选的参数是 `:ro` 或者 `:rw`，前者表示容器只读，后者表示容器对数据卷是可读可写的。默认情况下是可读可写的。

```
volumes_from:
- service_name
- service_name:ro
- container:container_name
- container:container_name:rw
```

23. 修改内核功能: `cap_add`, `cap_drop`

添加或删除容器的内核功能。详细信息在前面容器章节有讲解，此处不再赘述。

```
cap_add:
- ALL
```

```
cap_drop:
- NET_ADMIN
- SYS_ADMIN
```

24. 指定父级Cgroup: `cgroup_parent`

指定一个容器的父级 `cgroup`。

```
cgroup_parent: m-executor-abcd
```

25. 配置服务的设备映射: `devices`

设备映射列表。与 Docker client 的 `--device` 参数类似。

```
devices:
- "/dev/ttyUSB0:/dev/ttyUSB0"
```

26. 设置服务扩展: `extends`

`extends` 标签可以扩展另一个服务，扩展内容可以是来自当前文件，也可以是来自其他文件，相同服务的情况下，后来者会有选择地覆盖原有配置。

```
extends:
  file: common.yml
  service: webapp
```

用户可以在任何地方使用这个标签，只要标签内容包含 `file` 和 `service` 两个值就可以了。`file` 的值可以是相对或者绝对路径，如果不指定 `file` 的值，那么 Compose 会读取当前 YML 文件的信息。

更多的操作细节在 12.3.4 节会具体介绍。

27. 设置服务网络模式: `network_mode`

网络模式，与 Docker client 的 `--net` 参数类似，只是相对多了一个 `service:[service name]`

的格式。例如：

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

可以指定使用服务或者容器的网络。

28. 设置服务容器的网络：networks

加入指定网络，格式如下：

```
services:
  some-service:
    networks:
      - some-network
      - other-network
```

关于 `networks` 标签还有一个特别的子标签 `aliases`，这是一个用来设置服务别名的标签，例如：

```
services:
  some-service:
    networks:
      some-network:
        aliases:
          - alias1
          - alias3
      other-network:
        aliases:
          - alias2
```

相同的服务可以在不同的网络有不同的别名。

29. 其他标签

除此之外，还有这些标签：`cpu_shares`, `cpu_quota`, `cpuset`, `domainname`, `hostname`, `ipc`, `mac_address`, `mem_limit`, `memswap_limit`, `privileged`, `read_only`, `restart`, `shm_size`, `stdin_open`, `tty`, `user`, `working_dir`。

上面这些标签都是一个单值的标签，类似于使用 `docker run` 的效果。

```
cpu_shares: 73
cpu_quota: 50000
cpuset: 0,1

user: postgresql
working_dir: /code

domainname: foo.com
hostname: foo
ipc: host
mac_address: 02:42:ac:11:65:43

mem_limit: 1000000000
memswap_limit: 2000000000
privileged: true

restart: always
```

```
read_only: true
shm_size: 64M
stdin_open: true
tty: true
```

这些标签的使用方法与 Docker client 一样，这里不再重复。

12.3.3 网络配置

Compose 可以指定自定义网络，而不是使用默认的应用网络，这允许用户创建更复杂的拓扑结构和指定自定义网络驱动程序及选项。

以下面配置文件为例，proxy 服务位于项目的前端网络，app 服务位于中间，同时位于前端与后端网络，而 db 服务位于后端网络。

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
    image: postgres
    networks:
      - back

networks:
  front:
    // 使用自定义驱动
    driver: custom-driver-1
  back:
    // 使用自定义驱动以及可选参数
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

除了配置默认网络之外，还可以使用已经存在的网络，与前面 networks 标签类似，可以在 service 同级标签中设置 networks 覆盖全部服务容器。例如：

```
service:
  proxy:
    build: ./proxy
    .....
  app:
    build: ./app
    .....
  db:
    image: postgres
    .....
networks:
  default:
```

```
external:
  name: my-pre-existing-network
```

关于 `external` 的内容在 12.3.4 节会介绍，在没讲到集群内容之前，暂不会对 `overlay` 网络模型进行分析，在后面的章节再做讲解。

12.3.4 配置扩展

在上面的基础配置小节中，介绍到一个标签 `extends`，在 12.3.3 节的网络配置中还有一个 `external` 标签，这两个都属于 Compose 配置文件的一个扩展部分。

Compsoe 配置扩展有两个方法，一个是使用 `-f` 参数添加多个配置文件，在前面已经介绍过，第二种方法是使用 `extends` 标签，在 Compose 配置文件中可以使用该标签来扩展指定的服务。

关于第一个 `-f` 参数，其实 Compose 默认情况下不仅会读取 `docker-compose.yml` 文件，还会读取一个叫做 `docker-compose.override.yml` 的文件（如果存在），后者表示默认覆盖前者的配置，两者的名称在 `.env` 文件中可以定义。

例如：下面是 `docker-compose.yml` 的内容，在文件中定义了两个服务（默认不填写 `version` 会按照 Compsoe 版本自动补充），分别是 `web` 和 `db`。

```
web:
  image: example/my_web_app:latest
  links:
    - db
    - cache

db:
  image: postgres:latest

cache:
  image: redis:latest
```

然后，在同目录下的 `docker-compose.override.yml` 文件中定义了两个服务名称相同的服，但是服务内的定义并不相同，此时 `docker-compose.override.yml` 的配置会选择性地覆盖上面配置的定义。

```
web:
  build: .
  volumes:
    - './code'
  ports:
    - 8883:80
  environment:
    DEBUG: 'true'

db:
  command: '-d'
  ports:
    - 5432:5432

cache:
  ports:
    - 6379:6379
```

所谓选择性覆盖配置是指有冲突时以后面配置为准，无冲突时两者合并。例如，上面

两份配置文件选择性覆盖合并以后，实际内容如下：

```
web:
  build: .
  image: example/my_web_app:latest
  links:
    - db
    - cache
  volumes:
    - './code'
  ports:
    - 8883:80
  environment:
    DEBUG: 'true'

db:
  image: postgres:latest
  command: '-d'
  ports:
    - 5432:5432

cache:
  image: redis:latest
  ports:
    - 6379:6379
```

如果使用了 `-f` 参数指定多份配置文件，Compose 将不会读取 `docker-compose.override.yml` 文件。

而第二种办法就是使用 `extends` 标签，该标签原则上可以在配置文件的任何地方，但是一般把它放到服务定义子级中，例如：

```
web:
  extends:
    file: common-services.yml
    service: webapp
```

启动时 Compose 会从 `common-services.yml` 文件中读取扩展定义：

```
webapp:
  build: .
  ports:
    - "8000:8000"
  volumes:
    - "/data"
```

这时候启动只需要指定 `docker-compose.yml` 即可，不必使用 `-f` 再指定 `common-services.yml` 文件。

在同一个配置文件中还可以对同项目的服务做扩展，例如：

```
web:
  extends:
    file: common-services.yml
    service: webapp
  environment:
    - DEBUG=1
  cpu_shares: 5

important_web:
  extends: web
  cpu_shares: 10
```

在上面的配置中, `important_web` 基于 Web 服务扩展, 重新设置了 `cpu_shares` 的值, 而 Web 服务扩展自 `common-services.yml` 的 `webapp`。

最后是关于配置覆盖的注意要点, 前面提到过配置是选择覆盖的, 例如 `command` 标签, Compose 会把镜像 Dockerfile 中的定义与默认配置进行对比, 如果不同则用默认配置的值覆盖 Dockerfile 的值, 如果有扩展配置文件, 那么以扩展配置文件的值为准, 例如:

```
// Dockerfile 初始值
command: python app.py

// 本地配置文件的定义
command: python otherapp.py
```

```
// 实际执行的结果
command: python otherapp.py
```

这是单值标签的情况, 包含此种情况的标签还有 `image`、`mem_limit` 等。

在多重值标签中又有不同, 以 `expose` 标签为例:

```
// Dockerfile 初始值
expose:
  - "3000"
```

```
// 本地默认配置定义
expose:
  - "4000"
  - "5000"
```

```
// 实际运行时是前后两者合并, 因为两次定义没有产生冲突
expose:
  - "3000"
  - "4000"
  - "5000"
```

除了 `expose`, 还有一些标签也是这样, 例如 `ports`、`expose`、`external_links`、`dns`、`dns_search`、`tmpfs` 等。

除了合并定义的情况还有产生定义冲突的情况, 例如 `environment` 标签:

```
// Dockerfile 初始值
environment:
  - FOO=original
  - BAR=original
```

```
// 本地配置文件定义
environment:
  - BAR=local
  - BAZ=local
```

```
// 实际结果
environment:
  - FOO=original
  - BAR=local
  - BAZ=local
```

可以看到在合并的基础上, Compose 会把冲突的值处理, 以后来定义的值为准。

12.4 Compose 实战

在学习完前面的内容之后，相信读者已经掌握了基本的 Compose 操作，下面以 3 个综合的例子来熟悉 Compose 在实际环境中的运用。

12.4.1 部署 Django

Django 是一个开放源代码的 Web 应用框架，由 Python 写成。在 Python 社区乃至整个开源社区都是鼎鼎有名的框架。本节内容将实战 Compose 部署 Django 项目。

首先创建一个空的文件夹，文件名称默认就是项目名称，因此取名 web，然后在文件夹里面新建一个 Dockerfile 文件，用于构建 Django 应用的镜像。Dockerfile 的内容并不复杂，按照传统开发 Python 的方式，首先需要有一个 Python 基础镜像作为基础开发环境，这里选择 Python 2.7。

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
COPY requirements.txt /code/
WORKDIR /code
RUN pip install -r requirements.txt
ADD . /code/
```

保存 Dockerfile 之后，需要根据依赖编写 requirements.txt 文件，依赖不多，就只有以下两个：

```
Django
psycopg2
```

保存 requirements.txt 文件，然后一个 Django 镜像的基本材料就齐了。

但是现在还缺一份 docker-compose.yml 来编排整个过程，因此新建 docker-compose.yml 文件，定义两个服务，一个是数据库（db），另一个是 Django 应用（app），数据库选择 postgres，应用基于上面的 Dockerfile 构建。

```
version: '2'
services:
  db:
    image: postgres
  app:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

保存 docker-compose.yml 文件，现在已经完成了关于 Compose 的定义工作，接下来就是利用 Compose 生成一个 Django 项目。

```
$ docker-compose run app django-admin.py startproject compose_example .
```


还记得 `run` 命令的一次性特点吧，这里使用配置文件的 `app` 服务的定义，构建一个 Django 镜像，使用 `django-admin.py startproject compose_example` 会创建一个 Django 项目。执行之后查看项目文件夹，可以看到基本项目已经创建完成。

```
$ ls -l
drwxr-xr-x 2 root root compose_example
-rw-rw-r-- 1 user user docker-compose.yml
-rw-rw-r-- 1 user user Dockerfile
-rwxr-xr-x 1 root root manage.py
-rw-rw-r-- 1 user user requirements.txt
```

不过因为上面的 `Dockerfile` 中没有切换用户来执行创建项目的动作，默认使用了容器的 `root` 用户来创建，所以现在看到的 `compose_example` 项目是属于 `root` 用户的，如果想把项目目录切换为当前用户所在的用户组，可以使用 `chown` 切换（仅限 Linux 平台，其他平台没有这个步骤）：

```
$ sudo chown -R $USER:$USER .
```

现在项目已经创建，然后就需要配置数据库了，打开 `compose_example/settings.py` 文件，修改 `DATABASES = ...` 的内容如下：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

完成上面的步骤后，就已经完成全部运行前的工作了。

接着启动项目，`Compose` 会启动两个容器并连接它们：

```
$ docker-compose up
Starting web_db_1...
Starting web_app_1...
Attaching to web_db_1, web_app_1
.....
db_1 | PostgreSQL init process complete; ready for start up.
.....
db_1 | LOG: database system is ready to accept connections
db_1 | LOG: autovacuum launcher started
.....
web_1 | Django version 1.8.4, using settings 'compose_example.settings'
web_1 | Starting development server at http://0.0.0.0:8000/
web_1 | Quit the server with CONTROL-C.
```

打开浏览器，输入地址 `http://0.0.0.0:8000/`，不出意外的话，可以看到如图 12.2 所示的成功部署的页面。

It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

图 12.2 Django 成功部署

在这个例子中，以 Django 为例部署了一个与数据库连接的 Web 项目，把原本需要几条复杂的 `docker run` 命令才能部署的项目，简化为一句 `docker-compose up` 命令。

12.4.2 部署 Rails

在部署 Rails 应用之前，先和前面一样创建一个文件夹，在文件夹里创建 3 个文件，分别是 `Dockerfile`、`Gemfile` 和 `Gemfile.lock`，其中 `Dockerfile` 文件的内容如下：

```
FROM ruby:2.2.0
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev nodejs
RUN mkdir /myapp
WORKDIR /myapp
COPY Gemfile /myapp/Gemfile
COPY Gemfile.lock /myapp/Gemfile.lock
RUN bundle install
ADD . /myapp
```

而 `Gemfile` 文件的内容如下：

```
source 'https://rubygems.org'
gem 'rails', '4.2.0'
```

至于 `Gemfile.lock` 文件，只是一个空白文件。Linux 下使用 `touch Gemfile.lock` 可以创建空白文件。

完成上面的工作后就可以编写 `docker-compose.yml` 文件了。

```
version: '2'
services:
  db:
    image: postgres
  web:
    build: .
    command: bundle exec rails s -p 3000 -b '0.0.0.0'
    volumes:
      - ../myapp
    ports:
      - "3000:3000"
    depends_on:
      - db
```

在上面的配置文件中，`db` 服务使用已经构建并托管在仓库的 `postgres` 镜像，Web 服务则使用当前目录下的 `Dockerfile` 构建的镜像。

在启动项目之前，还需要使用 Rails 创建项目。

```
$ docker-compose run web rails new . --force --database=postgresql
--skip-bundle
```

执行这句命令之后，Compose 会先根据 `Dockerfile` 构建镜像，然后使用该镜像执行新建 Rails 项目的命令，也就是 `run` 后面的部分。

这样在当前目录下就会生成一个 Rails 项目。与上面的 Django 项目一样，Linux 下创建有个小问题，就是要修改文件所有权。

```
$ sudo chown -R $USER:$USER .
```

如果不想在每次新建项目时都修改所有权，可以在 `Dockerfile` 中使用 `USER` 指令指定用户，前提是需要使用 `RUN` 命令创建用户。此时打开 `Gemfile` 文件会发现该文件已经被覆

盖，可以去掉下面的注释（启用 Javascript runtime）：

```
gem 'therubyracer', platforms: :ruby
```

然后应用新的 Gemfile 时要重新构建镜像，使用如下命令：

```
$ docker-compose build
```

完成这些操作之后就可以配置数据库了，打开文件 config/database.yml，编辑内容如下：

```
development: &default
  adapter: postgresql
  encoding: unicode
  database: postgres
  pool: 5
  username: postgres
  password:
  host: db
```

```
test:
  <<: *default
  database: myapp_test
```

完成这些步骤后，就可以启动项目了。

```
$ docker-compose up
```

```
.....
myapp_web_1 | [2014-01-17 17:16:29] INFO WEBrick::HTTPServer#start: pid=1
port=3000
```

这个时候还需要使用命令创建数据库。

```
$ docker-compose run web rake db:create
```

如果项目运行正常，打开浏览器访问本地 3000 端口即可看到如图 12.3 所示的应用页面。

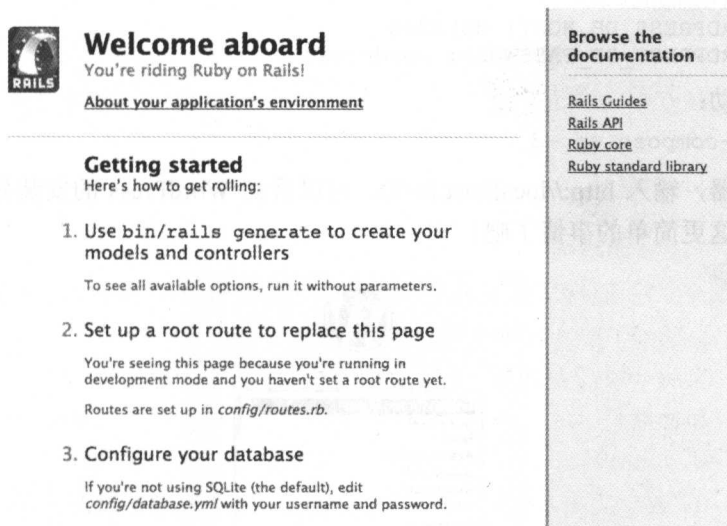


图 12.3 Rails 部署成功

12.4.3 部署 WordPress

WordPress 当之无愧是 PHP 语言的“杀手”级应用，它应用之广泛是其他 CMS 望尘

莫及的。相信读者中也不乏有 WordPress 的用户。还记得第一次手动部署 WordPress 站点时的经历吗？LNMP/LAMP 的安裝配置、WordPress 的安裝等烦琐工作让不少新手一筹莫展。

现在有了 Docker，一切都变得简单了，用户甚至不需要知道镜像里有什么，没有安裝，甚至没有配置，只是一个启动命令就可以完成全部操作。

要部署一个 WordPress，依旧要先创建一个空文件夹，在该文件夹里新建 docker-compose.yml，内容如下：

```
version: '2'

services:
  db:
    image: mysql:5.7
    volumes:
      - "../.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    links:
      - db
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
```

保存，启动：

```
$ docker-compose up -d
```

打开浏览器，输入 <http://localhost:8000>，可以看到 WordPress 的安裝界面如图 12.4 所示，再没有比这更简单的事情了吧！

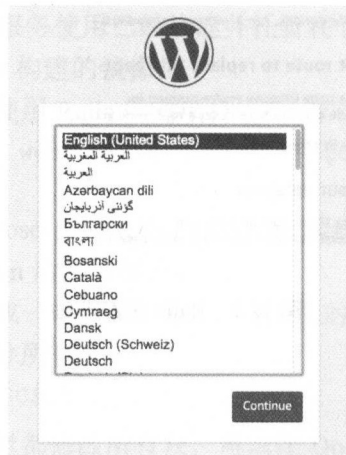


图 12.4 WordPress 部署成功

那么上面的 `docker-compose.yml` 文件到底做了什么呢？可以看到配置文件中定义了两个服务，一个是 `db`，另一个是 `wordpress`，两个服务基于现成的镜像（数据库使用 `mysql:5.7`，`wordpress` 在 Docker Hub 有官方镜像）因此启动速度很快，没有构建过程。

数据库使用了一个数据卷来保存数据，宿主机目录是 `./data/db`，数据库文件被保存在这里，`environment` 标签定义了多个数据库变量。`Wordpress` 服务连接到数据库中，映射容器的 80 端口到本地的 8000 端口。

更详细的 `WordPress` 镜像使用方法可以看 Docker Hub 的 `WordPress` 页面 https://hub.docker.com/r/_/wordpress/。

12.5 本章小结

本章围绕 `Docker Compose` 这个常用的编排工具进行详细的讲解与实战，包括对 `Docker Compose` 的命令解释与应用，还详细解释了 `docker-compose.yml` 文件的编写规则，在后面的章节中会大量使用到这些知识。

最后以 3 个著名的框架、应用为实战例子，使用 `Docker Compose` 快速部署，体验 `Docker Compose` 在操作管理容器方面的优势。

第 13 章 Web 服务器与应用

Web 服务器是网络应用中最常见、最重要的一个部分。本章将会重点介绍使用 Docker 来运行常见的 Web 服务器。首先介绍官方镜像，有哪些版本，版本有哪些差异。然后运行官方镜像，这些镜像与传统 Web 服务器相比有哪些参数和功能。再分别基于 Alpine 和 Ubuntu 构建 Web 服务器应用。最后介绍第三方的优质镜像。

通过本章的学习，读者可以根据自己的需要自由定制自己的 Web 服务器。

13.1 Apache 服务器

说到 Web 服务器的 Apache 通常是指 Apache HTTP Server。这是 Apache 软件基金会有一个开放源代码的网页服务器软件，可以在大多数计算机操作系统中运行，由于其跨平台 and 安全性，被广泛使用，是最流行的 Web 服务器软件之一。

下面先从认识官方镜像开始，然后通过 Dockerfile 来构建自己的 Apache 服务器镜像。

13.1.1 官方镜像

一般来说，官方镜像都是存放在 library (<https://hub.docker.com/u/library>) 组织中，用户可以通过下面这些格式访问官方镜像。

```
https://hub.docker.com/_/镜像名称/  
https://hub.docker.com/r/_/镜像名称/  
https://hub.docker.com/r/library/镜像名称/
```

虽然 Apache HTTP 服务器通常称为 Apache Web 服务器，但是在更正式的命名上，我们称为 httpd。Apache 官方镜像的地址是：

```
https://hub.docker.com/r/library/httpd/
```

而不是像下面的地址。

```
https://hub.docker.com/r/library/apache/
```

在镜像介绍区域中，可以看到目前 Apache 的版本 (tag) 和对应的 Dockerfile，如图 13.1 所示。

- 2.2.31, 2.2 (2.2/Dockerfile)
- 2.2.31-alpine, 2.2-alpine (2.2/alpine/Dockerfile)
- 2.4.23, 2.4, 2, latest (2.4/Dockerfile)
- 2.4.23-alpine, 2.4-alpine, 2-alpine, alpine (2.4/alpine/Dockerfile)

图 13.1 目前 Apache 的版本

在 Docker Hub 中,这些由官方维护的镜像一般都经过一定的审核,用户可以放心使用。官方提供的镜像中一般包含两个不同的版本,分别基于 Debian / Ubuntu 和 Alpine 构建。

通过 Dockerfile 我们可以知道镜像里面有些什么,然后根据自己的需要进行修改。在 Github 上可以查看这些镜像的 Dockerfile。下面以 httpd 2.4 的 Dockerfile 为例,学习 Apache 镜像的构建。

```
# 基于 Debian 构建
FROM debian:jessie

# 这一句看情况是否需要添加,出现没有这个用户或用户组时加入下面这行,以分配 ID
#RUN groupadd -r www-data && useradd -r --create-home -g www-data www-data

# 创建工作目录并且设置用户权限
ENV HTTPD_PREFIX /usr/local/apache2
ENV PATH $HTTPD_PREFIX/bin:$PATH
RUN mkdir -p "$HTTPD_PREFIX" \
    && chown www-data:www-data "$HTTPD_PREFIX"
WORKDIR $HTTPD_PREFIX

# install httpd runtime dependencies
# 安装依赖,具体文档参考下面链接
# https://httpd.apache.org/docs/2.4/install.html#requirements
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        libapr1 \
        libaprutil1 \
        libaprutil1-ldap \
        libapr1-dev \
        libaprutil1-dev \
        libpcre++0 \
        libssl1.0.0 \
    && rm -r /var/lib/apt/lists/*

# 定义版本变量和 SHA1 值,确保传输过程安全
ENV HTTPD_VERSION 2.4.23
ENV HTTPD_SHA1 5101be34ac4a509b245adb70a56690a84fcc4e7f
ENV HTTPD_BZ2_URL https://www.apache.org/dist/httpd/httpd-$HTTPD_VERSION.
tar.bz2

# 使用设置 buildDeps 变量的方式,方便后面卸载这些依赖
RUN set -x \
    && buildDeps=' \
        bzip2 \
        ca-certificates \
        gcc \
        libpcre++-dev \
        libssl-dev \
        make \
        wget \
    ' \
    && apt-get update \
    && apt-get install -y --no-install-recommends $buildDeps \
    && rm -r /var/lib/apt/lists/* \
    && wget -O httpd.tar.bz2 "$HTTPD_BZ2_URL" \
    && echo "$HTTPD_SHA1 *httpd.tar.bz2" | shasum -c - \
# see https://httpd.apache.org/download.cgi#verify
    && wget -O httpd.tar.bz2.asc "$HTTPD_BZ2_URL.asc" \
```



```

    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys A93D62ECC3
C8EAl2DB220EC934EA76E6791485A8 \
    && gpg --batch --verify httpd.tar.bz2.asc httpd.tar.bz2 \
    && rm -r "$GNUPGHOME" httpd.tar.bz2.asc \
    \
    && mkdir -p src \
    && tar -xvf httpd.tar.bz2 -C src --strip-components=1 \
    && rm httpd.tar.bz2 \
    && cd src \
    \
    && ./configure \
        --prefix="$HTTDP_PREFIX" \
        --enable-mods-shared=reallyall \
    && make -j "$(nproc)" \
    && make install \
    \
    && cd .. \
    && rm -r src \
    \
    && sed -ri \
        -e 's!^\(s*CustomLog\)s+\S+!l /proc/self/fd/1!g' \
        -e 's!^\(s*ErrorLog\)s+\S+!l /proc/self/fd/2!g' \
        "$HTTDP_PREFIX/conf/httpd.conf" \
    \
    && apt-get purge -y --auto-remove $buildDeps
# 上面用“\”符号连接命令，跨行显示有助于阅读
# 启动命令脚本
COPY httpd-foreground /usr/local/bin/

EXPOSE 80
CMD ["httpd-foreground"]

```

httpd-foreground 内容:

```

#!/bin/sh
set -e
# Apache gets grumpy about PID files pre-existing
rm -f /usr/local/apache2/logs/httpd.pid
exec httpd -DFOREGROUND

```

Apache 的 Dockerfile 还是非常容易理解的，下面来看如何运行这个镜像，以及要注意的细节。

13.1.2 运行官方镜像

下面使用 Docker 真正运行一下 Apache。为了方便用户使用，这些镜像一般都设置了很多非常方便的操作，比如在 Apache 上，可以直接使用 Apache 镜像做基础镜像构建用户的 Web 应用。

```

FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/

```

其中，./public-html/为用户当前目录下的文件夹。直接执行：

```
$ docker build -t my-apache2 .
```

镜像构建后使用 `docker run` 命令运行时会自动启动 Web 服务器，这样 Web 应用就启动好了。但是这样显然还是挺麻烦的。我们还可以通过数据卷直接挂载 Web 应用到容器中。

Apache 镜像一样预留了对应的数据卷：

```
$ docker run -dit --name my-apache-app -v "$PWD":/usr/local/apache2/htdocs/
httpd:2.4
```

这样,就可以把本地的 Web 应用运行在容器中,省去了传统 Web 应用部署一个 Apache 的步骤。

在官方镜像中,默认使用了配置文件,用户可以使用自己自定义的 Apache 配置文件。默认的配置文件的存放在 `/usr/local/apache2/conf/httpd.conf` 中。如果要使用自己的配置文件同样有两种方式,一种是使用 Dockerfile 再构建一个镜像,在 Dockerfile 中加入:

```
FROM httpd:2.4
.....
COPY ./my-httpd.conf /usr/local/apache2/conf/httpd.conf
.....
```

然后构建镜像,就会把默认配置替换为你的配置。

每次修改配置都要重新构建镜像显然很麻烦,我们还可以使用数据卷的方式让容器读取配置文件。假设配置文件地址如下:

```
/usr/local/apache2/conf/httpd.conf
```

那么在启动容器时可以使用:

```
-v /YOUR/PATH/httpd.conf:/usr/local/apache2/conf/httpd.conf:ro
```

这样的参数,这样容器启动时会读取宿主机的配置文件,而不用重新构建镜像。(:ro 表示容器“只读”)

在 Apache 中启用 SSL 一样只需 COPY crt 文件或者挂载 crt 文件。不管是哪种方式,都需要在配置文件(默认是在 `/usr/local/apache2/conf/httpd.conf`)中修改。具体修改和传统 Apache 服务器一样,去掉注释“`#Include conf/extra/httpd-ssl.conf`”。

13.1.3 基于 Ubuntu 构建 Apache 镜像

13.1.2 节讲了官方镜像。本节构建一个特殊的,适合自己用的 Apache 镜像。

因为从本章开始接下来的两章都是关于镜像实战的内容,所以首先创建一个文件夹来统一存放后续学习所创建的文件,这里命名为 `dockerfiles`。然后在 `dockerfiles` 里面再创建目录 `apache`,用于存放本节后面创建的相关信息。

```
$ mkdir -p ~/dockerfiles/apache/www \
~/dockerfiles/apache/logs \
~/dockerfiles/apache/conf
```

解释一下目录的用途如下:

- `www` 目录将映射为 `apache` 容器配置的应用程序目录;
- `logs` 目录将映射为 `apache` 容器的日志目录;
- `conf` 目录里的配置文件将映射为 `apache` 容器的配置文件。

进入创建的 `apache` 目录,创建 Dockerfile,和官方的版本不一样,这次我们从软件仓库直接安装 Apache。

```
FROM debian:jessie
MAINTAINER Zuo Lan <i@zuolan.me>
```

```
# 安装并清理
RUN apt-get update && apt-get -y install apache2 \
    && apt-get autoclean -y && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# 设置用户组和用户, 并设置日志文件夹
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

# 开启 SSL
RUN /usr/sbin/a2ensite default-ssl
RUN /usr/sbin/a2enmod ssl

EXPOSE 80
EXPOSE 443

# 设置启动命令
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

保存文件然后构建:

```
$ docker run build -t apache .
```

在这个镜像中:

- /var/www/html 是容器内的网站根目录 (也是 Apache2 默认的根目录);
- /var/log/apache2 是容器内的日志目录;
- /etc/ssl 是容器内的 SSL 密钥存放目录;
- /etc/apache2/ 是容器内部 Apache 的配置目录。

得知这些信息后, 相信读者已经知道应该怎么使用这个镜像了。例如, 下面即可启动一个 Apache 服务器:

```
$ docker run -d -p 80:80 \
-v ~/dockerfiles/apache/www:/var/www/html \
-v ~/dockerfiles/apache/logs:/var/log/apache2 \
-v ~/dockerfiles/apache/apache2.conf:/etc/apache2/apache2.conf \
apache
```

13.1.4 基于 Alpine 构建 Apache 镜像

在上面构建的镜像中可以看到, 镜像体积相对还是算小的, 但对于 Apache 服务器镜像来说显然还可以更精简。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
apache	latest	0931d9b704cc	17 minutes ago	183.8 MB
httpd	latest	462fca6e7913	5 weeks ago	195.4 MB

接下来将使用 Alpine 构建 Apache 镜像。与上面的基于 Debian 的镜像不同, 基于 Alpine 的镜像在生产环境中可能会遇到一些“意外”。尽管 Alpine 已经乘着 Docker 的东风飞速发展, 但是对于 Alpine 的稳定性和兼容性还是要留个心眼。所以在构建 Alpine 镜像时除了要精简, 还要确保可扩展性, 特别是对于模块化的程序。

根据惯例还是先进入文件夹 dockerfiles/apache, 然后创建一个文件夹 alpine, 用于存放 apache:alpine 版本的 Dockerfile, 最后新建 Dockerfile (Github 地址为 <https://github.com/nimmis/docker-alpine-apache>)。

```
FROM nimmis/alpine-micro
COPY root/. /

RUN apk update && apk upgrade && \
  # 安装 Apache
  apk add apache2 libxml2-dev apache2-utils && \
  mkdir /web/ && chown -R apache.www-data /web && \
  # 配置 Apache
  sed -i 's#^DocumentRoot".*#DocumentRoot "/web/html"#g' /etc/apache2/
  httpd.conf && \
  sed -i 's#AllowOverride none#AllowOverride All#' /etc/apache2/
  httpd.conf && \
  sed -i 's#^ServerRoot .*#ServerRoot /web#g' /etc/apache2/httpd.conf && \
  sed -i 's#^#ServerName.*#ServerName webproxy/' /etc/apache2/httpd.conf && \
  sed -i 's#^IncludeOptional /etc/apache2/conf#IncludeOptional /web/
  config/conf#g' /etc/apache2/httpd.conf && \
  sed -i 's#PidFile"/run/*#Pidfile "/web/run/httpd.pid"#g' /etc/apache2/
  conf.d/mpm.conf && \
  sed -i 's#Directory"/var/www/localhost/htdocs.*#Directory "/web/html"
  >#g' /etc/apache2/httpd.conf && \
  sed -i 's#Directory"/var/www/localhost/cgi-bin.*#Directory "/web/cgi-
  bin" >#g' /etc/apache2/httpd.conf && \
  sed -i 's#/var/log/apache2/#/web/logs/#g' /etc/logrotate.d/apache2 && \
  sed -i 's/Options Indexes/Options /g' /etc/apache2/httpd.conf && \
  # 清理缓存
  rm -rf /var/cache/apk/*

# 设置 Web 目录为数据卷
VOLUME /web
EXPOSE 80 443
```

这份 Dockerfile 中常用目录的位置如表 13.1 所示。镜像作者修改了 Apache 的部分配置内容，使得镜像更容易操作与控制。

表 13.1 镜像常用目录

目 录	Function
/web/html	Web 根目录
/web/cgi-bin	Cgi bin 目录
/web/config	Apache 配置目录
/web/logs	Apache 日志目录
/web/internal	内部页面、错误页面等

本镜像源码在 13.1.5 节有链接，读者可以阅读完整的 Dockerfile 源码。

13.1.5 第三方优质镜像

前面说过，除了官方镜像，有些开发者构建的镜像也非常出色，不妨参考他们的 Dockerfile 来构建自己的镜像，或者直接使用他们的镜像。如表 13.2 所示为比较推荐的 Apache 镜像。

表 13.2 推荐镜像

镜 像 地 址	推 荐 理 由
https://hub.docker.com/r/webdevops/apache/	Webdevops 的官方镜像，有生产环境经验
https://hub.docker.com/r/bitnami/apache/	比官方镜像操作更灵活
https://github.com/nimmis/docker-alpine-apache	使用 Alpine 构建而来，体积小，功能齐全

更多的好镜像当然是要自己去发现，可以通过 Docker Hub 搜索来获取信息。

13.2 Nginx 服务器

Nginx 和 Apache 一样是一种 Web 服务器，与 Apache 相比，它内存占用更小、稳定性更高。Nginx 不采用每客户机一线程的设计模型，而是充分使用异步逻辑，削减了上下文调度开销，所以并发服务能力更强，整体采用模块化设计，有丰富的模块库和第三方模块库，配置灵活。

13.2.1 官方镜像介绍

在官方镜像地址可以看到 Nginx 的版本，如图 13.2 所示。

- 1.11.3, mainline, 1, 1.11, latest (mainline/jessie/Dockerfile)
- 1.11.3-alpine, mainline-alpine, 1-alpine, 1.11-alpine, alpine (mainline/alpine/Dockerfile)
- 1.10.1, stable, 1.10 (stable/jessie/Dockerfile)
- 1.10.1-alpine, stable-alpine, 1.10-alpine (stable/alpine/Dockerfile)

图 13.2 Nginx 版本

可以看到，官方镜像一样提供了基于 Debian 和 Alpine 构建的两个版本。这两个版本的功能大致相同，但是依赖版本因社区规则不同而有差异。例如，默认的 Nginx:latest 和 Nginx:alpine 的 openssl 版本就不一样，如果你的网站需要使用 http/2，那么使用目前的 latest 镜像还需要手动编译 Nginx 镜像（截至 2016 年 09 月 01 号），而使用 alpine 版本的镜像则可以直接开启 http/2。除了这些之外，在使用上也有细微差别，如 alpine 镜像内没有 Bash 工具，一些容器内执行的脚本就需要做相应修改了。

最重要的是，使用基于 Debian 的 Nginx 镜像做基础镜像构建新的镜像时，难度会相对较低点。因为 Debian 大部分读者都会有所接触。而使用 alpine 来构建镜像除了操作上的陌生外，还存在 Alpine 软件仓库缺乏部分依赖软件的问题，需要手动解决，这是一件有难度的事情。与 Debian 版本的 Nginx 镜像相比，alpine 的 Nginx 镜像小得多，非常适合一些场景（对镜像大小有要求）的应用。

13.2.2 运行官方镜像

目前 Docker Hub 官方提供的 Nginx 镜像与传统的 Nginx 服务器功能上无差异，但操作

上有区别。而且镜像里的 Nginx 是编译版本，对于模块扩展等需求有些限制。下面先从基本的用法开始。

直接运行 Nginx 容器作为静态服务器，这大概是 Nginx 最简单的应用场景之一了。镜像预留了相应的 VOLUME，所以直接执行 `docker run -v` 并指定目录即可运行一个静态服务器。如下：

```
$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro
-d nginx
```

上面一句中，`:ro` 表示容器对这个文件夹只读，这样有利于保护数据安全，防止因为容器内部错误而删改数据内容。

另外一种情况是可以使用 Dockerfile 来构建静态服务器。

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

修改路径直接执行 `docker build` 即可。修改 80 端口只需要在 `docker run -p` 后面设定参数，如 `-p 8080:80` 这样。

很多时候自己运行 Nginx 都需要使用自己的配置文件。使用 Nginx 镜像时也不例外，官方也提供了相应的方法。

```
$ docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/nginx.conf:ro
-d nginx
```

这里依旧使用了 `:ro` 的权限，保证了配置文件容器不可改写。

老规矩，还是可以通过 Dockerfile 构建而不用在启动命令中添加参数。

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

获取容器默认的配置可以通过 `docker cp` 获得。

```
$ docker cp some-nginx:/etc/nginx/nginx.conf /some/nginx.conf
```

但是仅仅这样有时并不能满足 Nginx 配置要求，比如有多个站点，每个站点独立一份配置文件时就不能使用上面的方法了。所以需要使用 Nginx 的 `conf.d` 目录：

```
$ docker run --name some-nginx -v /path/nginx.d:/etc/nginx/conf.d:ro -d
nginx
```

通过上面的命令运行 Nginx 容器显然会使 `docker run` 命令很冗长，所以可以通过 `docker-compose` 来启动 Nginx。

```
image: nginx
volumes:
- ./mysite.template:/etc/nginx/conf.d/mysite.template
ports:
- "8080:80"
environment:
- NGINX_HOST=foobar.com
- NGINX_PORT=80
command: /bin/bash -c "envsubst < /etc/nginx/conf.d/mysite.template >
/etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
```

在配置文件中可以使用 `listen ${NGINX_PORT}`；这样的方式定义，这样的话修改端口只需要在 `docker-compose.yml` 中修改。

以上只对 Nginx 镜像的简单使用，更复杂的使用需要读者自己动手定制。13.2.3 节将

使用两种不同的方式定制 Nginx 镜像。

13.2.3 构建 Nginx 镜像

1. 是基于 Debian 构建 Nginx 镜像

```
$ mkdir -p ~/nginx/www ~/nginx/logs ~/nginx/conf
```

- www 目录将映射为 Nginx 容器配置的虚拟目录;
- logs 目录将映射为 Nginx 容器的日志目录;
- conf 目录里的配置文件将映射为 Nginx 容器的配置文件。

创建 Nginx 文件夹, 分别存放接下来要用的文件, 然后新建 Dockerfile。

```
FROM debian:jessie
MAINTAINER Zuo Lan "i@zuolan.me"

ENV NGINX_VERSION 1.10.1-1~jessie

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8F
BC641079A6ABABF5BD827BD9BF62 \
    && echo "deb http://nginx.org/packages/debian/ jessie nginx" >>
/etc/apt/sources.list \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y \
        ca-certificates \
        nginx=${NGINX_VERSION} \
        nginx-module-xslt \
        nginx-module-geoip \
        nginx-module-image-filter \
        nginx-module-perl \
        nginx-module-njs \
        gettext-base \
    && rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

上面的 nginx-module-* 可以修改或添加用户需要的模块 (<http://nginx.org/en/docs/>), 然后构建 Nginx 如下:

```
$ docker build -t nginx .
```

查看 Nginx 镜像:

```
$ docker images nginx
REPOSITORY    TAG        IMAGE ID      CREATED      SIZE
nginx         latest    2d5bbd91e13c  3 days ago  182.8 MB
```

如果不知道默认配置, 则可以使用:

```
$ docker cp some-nginx:/etc/nginx/nginx.conf /some/nginx.conf
```

这样就可以拿到默认的配置文​​件, 然后进行修改, 再挂载上去。

2. 基于Alpine构建Nginx镜像

参考源码 <https://github.com/lscience/docker-nginx>。

在下面的这份 Dockerfile 中，从 Nginx 官网下载并自动编译对应的版本。用户可以自定义 Dockerfile 内容以达到自己的需求。

```
FROM alpine:3.2

# 设置 Nginx 版本
ENV NGINX_VERSION=1.10.1 NGINX_HOME=/usr/share/nginx

# 从源码编译 Nginx
RUN apk update && apk add curl openssl-dev pcre-dev zlib-dev wget build-base
RUN curl -Ls http://nginx.org/download/nginx-${NGINX_VERSION}.tar.gz | tar
-xz -C /tmp && \
  cd /tmp/nginx-${NGINX_VERSION} && \
  ./configure \
    --with-http_ssl_module \
    --with-http_gzip_static_module \
    --prefix=${NGINX_HOME} \
    --conf-path=/etc/nginx/nginx.conf \
    --http-log-path=/var/log/nginx/access.log \
    --error-log-path=/var/log/nginx/error.log \
    --pid-path=/var/run/nginx.pid \
    --sbin-path=/usr/sbin/nginx && \
  make && \
  make install && \
  apk del build-base && mkdir -p /etc/nginx/conf.d && \
  rm -rf /tmp/*

# 请求和错误日志转发给 Docker 日志收集器
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

# 添加 Nginx 默认配置
#ADD etc /etc

# 暴露端口
EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

执行构建如下：

```
$ docker build -t user/nginx:alpine .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM alpine:3.2
--> 4933271a21f1
Step 2 : ENV NGINX_VERSION 1.10.1 NGINX_HOME /usr/share/nginx
--> Using cache
--> 74da3111415a
Step 3 : RUN apk update && apk add curl openssl-dev pcre-dev zlib-dev wget
build-base
--> Using cache
--> d2bc7ca6e3c9
Step 4 : RUN curl -Ls http://nginx.org/download/ nginx-${NGINX_VERSION}.
tar.gz | tar -xz -C /tmp && cd /tmp/nginx-${NGINX_VERSION} && .
/configure --with-http_ssl_module
--with-http_gzip_static_module --prefix=${NGINX_HOME}
```

```
--conf-path=/etc/nginx/nginx.conf      --http-log-path=/var/log/nginx/access.log
--error-log-path=/var/log/nginx/error.log  --pid-path=/var/run/nginx.pid
--sbin-path=/usr/sbin/nginx &&      make &&      make install &&      apk del
build-base && mkdir -p /etc/nginx/conf.d &&      rm -rf /tmp/*
---> Using cache
---> bf7a824ad8b3
Step 5 : RUN ln -sf /dev/stdout /var/log/nginx/access.log
---> Using cache
---> 20f869a9a2f1
Step 6 : RUN ln -sf /dev/stderr /var/log/nginx/error.log
---> Using cache
---> 98fc56ad2c54
Step 7 : EXPOSE 80 443
---> Using cache
---> aed71612babd
Step 8 : CMD nginx -g daemon off;
---> Using cache
---> 061125d508f0
Successfully built 061125d508f0
```

构建成功后执行 `docker run` 命令测试一下。不出意外的话，可以看到如图 13.3 所示的界面。

```
$ docker run --rm -p 80:80 user/nginx:alpine
```



图 13.3 Alpine Nginx 界面

13.2.4 第三方镜像推荐

如表 13.3 所示为目前 Docker Hub 上比较流行的第三方 Nginx 镜像。用户可以根据自己的需要选择适合的镜像。

表 13.3 Nginx推荐镜像

推 荐 镜 像	推 荐 理 由
https://hub.docker.com/r/webdevops/nginx/	Webdevops 的镜像，一个系列，有成熟经验
https://hub.docker.com/r/jwilder/nginx-proxy/	非常著名的镜像，智能反代理
https://hub.docker.com/r/lscience/alpine/	基于 Alpine，功能完全

13.3 Tomcat 服务器

Tomcat 是 Apache 软件基金会（Apache Software Foundation）的 Jakarta 项目中的一个核心项目。由于有 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总是能在 Tomcat 中得到

体现。因为 Tomcat 技术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱，并得到了部分软件开发商的认可，成为目前比较流行的 Web 应用服务器。

13.3.1 官方镜像介绍

Tomcat 在 Docker Hub 上一共有几十个标签，原因是 JRE 的版本与 Tomcat 的版本很多。例如，基于 JRE 6 构建的 Tomcat 6/7/8/9；基于 JRE 7 构建的 Tomcat 也有 6/7/8/9 等几个版本；还有 JRE 8 构建的 Tomcat 也是有几个版本。其中还不包括基于 Debian 和基于 Alpine 版本的分类。因此 Tomcat 的标签非常多，用户根据自己项目的特点选择适合的版本。一般来说选择稳定的版本就行，如图 13.4 所示为其 Logo。

如果不知道该使用哪个镜像也不要紧。Tomcat 镜像比较特别，它与 Apache 和 Nginx 不同。Tomcat 镜像自带了 JRE 运行环境，因此相当于 Apache+PHP 或者 Nginx+PHP 一样。使用 Tomcat 镜像可以直接挂数据卷运行 Web 应用，因此一个镜像运行不了不妨换一个，相当方便（从另一个角度看，要换来换去也是挺麻烦的）。

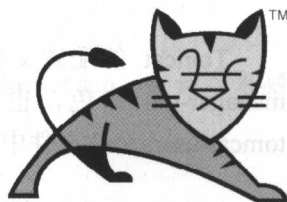


图 13.4 Tomcat Logo

基于 Alpine 构建的镜像，因为受制于 JRE 的特点，有可能不是很稳定，在后面编程语言的章节我们会亲自构建一个基于 Alpine 的 Java 基础镜像来体验精简的 JRE 环境。

13.3.2 运行官方镜像

要了解一个镜像如何运行，通常看最后一句 Dockerfile 执行了什么就大概明白了。

```
CMD ["catalina.sh", "run"]
```

看起来与普通环境下使用 Tomcat 差不多，也是执行 catalina.sh 文件。Tomcat 的 latest 标签是基于 JRE 7 构建的 Tomcat 8 镜像。本文以此为例，看看 Dockerfile 文件的内容（节选）。

```
FROM openjdk:7-jre
ENV CATALINA_HOME /usr/local/tomcat
WORKDIR $CATALINA_HOME
EXPOSE 8080
```

根据 CMD 和 WORKDIR 已经得知该镜像的 Tomcat 目录分布，EXPOSE 也标明了，所以现在已经知道如何运行一个 Tomcat 镜像了。最简单的直接运行：

```
$ docker run -it --rm tomcat:8.0
```

现在可以在浏览器中访问 <http://container-ip:8080> 了。如果一切正常，将会看到欢迎界面。注意，此时访问的是容器 IP，这是一个虚拟网络，在其他机器是无法访问的，此时也不能通过宿主机上的 8080 端口访问。

如果想从宿主机的 8080 端口访问，可以映射到宿主机：

```
$ docker run -it --rm -p 8080:8080 tomcat:8.0
```

现在可以打开浏览器，通过 `http://localhost:8888` 或者 `http://host-ip:8888` 访问应用。

Tomcat 7 和 Tomcat 8 的默认环境变量如下：

```
CATALINA_BASE: /usr/local/tomcat
CATALINA_HOME: /usr/local/tomcat
CATALINA_TMPDIR: /usr/local/tomcat/temp
JRE_HOME: /usr
CLASSPATH: /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar
```

Tomcat 6 的默认环境变量如下：

```
CATALINA_BASE: /usr/local/tomcat
CATALINA_HOME: /usr/local/tomcat
CATALINA_TMPDIR: /usr/local/tomcat/temp
JRE_HOME: /usr
CLASSPATH: /usr/local/tomcat/bin/bootstrap.jar
```

Tomcat 的配置文件可以放置在 `/usr/local/tomcat/conf/` 目录中。默认情况下没有设置 `manager-gui` 角色，也就不能通过 `/manager/html` 操作 Web 容器。如果需要，可以在 `tomcat-users.xml` 文件中设置。

13.3.3 构建 Tomcat 镜像

和上面一样，先基于 Ubuntu 构建一个 Tomcat 镜像。一个 Tomcat 镜像必然要有 Tomcat 软件与 JRE 环境，本节内容源代码参考 <https://github.com/tutumcloud/tomcat>。Tutum Cloud 原本是一家以 Docker 容器为服务内容的初创公司，后来被 Docker 公司收购，推出了 Docker Cloud。虽然公司被收购了，但是留下了不少优质的 Dockerfile，其中就有 Tomcat 的镜像。先看 Dockerfile：

```
FROM openjdk:7-jre

RUN apt-get update && \

# 安装 wget，后面需要下载文件
# 安装 pwgen，后面需要自动生成密码
# 安装 ca-certificates，网络应用不能忘了它
# 安装 libtcnative，Apache Tomcat Native
apt-get install -yq --no-install-recommends wget pwgen ca-certificates
&& \
    apt-get install -yq --no-install-recommends libtcnative-1 && \
    apt-get clean && \
rm -rf /var/lib/apt/lists/*

# 设置环境变量，方便统一做出调整，例如调整版本号
ENV TOMCAT_MAJOR_VERSION 8
ENV TOMCAT_MINOR_VERSION 8.0.22
ENV CATALINA_HOME /tomcat
ENV JAVA_OPTS ""

# 安装 Tomcat
RUN wget -q https://archive.apache.org/dist/tomcat/tomcat-${TOMCAT_MAJOR_VERSION}/v${TOMCAT_MINOR_VERSION}/bin/apache-tomcat-${TOMCAT_MINOR_VERSION}.tar.gz && \
    wget -qO- https://archive.apache.org/dist/tomcat/tomcat-${TOMCAT_MAJOR_VERSION}/v${TOMCAT_MINOR_VERSION}/bin/apache-tomcat-${TOMCAT_MINOR_VERSION}
```

```
ION}.tar.gz.md5 | md5sum -c - && \

# 下载完成并校验之后,解压到相应目录完成安装
tar xzf apache-tomcat-*.tar.gz && \
rm apache-tomcat-*.tar.gz && \
mv apache-tomcat* tomcat && \
rm -fr tomcat/webapps/examples && \
rm -fr tomcat/webapps/docs

# 添加生成用户的脚本,在启动的时候自动生成用户(如果不指定的话)
ADD create_tomcat_admin_user.sh /create_tomcat_admin_user.sh
ADD run.sh /run.sh
RUN chmod +x /*.sh

EXPOSE 8080
CMD ["/run.sh"]
```

关于 `create_tomcat_admin_user.sh` 与 `run.sh` 两个脚本,可以在上面的地址中找到源码,因此这里就不再给出了。使用 `docker build` 命令构建之后,可使用 `docker run -rm` 命令来测试,可以看到类似下面的内容:

```
$ docker run --rm 8080:8080 user/tomcat
=====
You can now connect to this Tomcat server using:
    admin:bluKcRK3r6SF
Please remember to change the above password as soon as possible!
=====
```

打开浏览器输入 `http://127.0.0.1:8080/manager/html`; 或者输入 `http://127.0.0.1:8080/host-manager/html`; 登录管理 Web 应用。

上面的启动命令中可以设置如下密码:

```
-e TOMCAT_PASS="mypass"
```

接下来基于 Alpine 构建一个 Tomcat 镜像。Alpine 安装 JRE 并不是一件轻松的事情。幸运的是社区已经有大量基于 Alpine 的 Java 镜像。因此我们也基于这些成果构建一个 Tomcat 镜像。有了 JRE, 安装 Tomcat 就是水到渠成的事情了。代码参考地址为 <https://github.com/davidcaste/docker-alpine-tomcat>。

```
FROM davidcaste/alpine-java-unlimited-jce:jre8
# 设置环境变量,使用“\”符号连接可以跨行显示,方便阅读
ENV TOMCAT_MAJOR=8 \
    TOMCAT_VERSION=8.5.3 \
    TOMCAT_HOME=/opt/tomcat \
    CATALINA_HOME=/opt/tomcat \
    CATALINA_OUT=/dev/null

# 使用 curl 下载 Tomcat 压缩包
RUN apk upgrade --update && \
    apk add --update curl && \
    curl -jksSL -o /tmp/apache-tomcat.tar.gz http://archive.apache.org/dist/tomcat/tomcat-${TOMCAT_MAJOR}/v${TOMCAT_VERSION}/bin/apache-tomcat-${TOMCAT_VERSION}.tar.gz && \

# 解压并设置 Tomcat
gunzip /tmp/apache-tomcat.tar.gz && \
tar -C /opt -xf /tmp/apache-tomcat.tar && \
ln -s /opt/apache-tomcat-${TOMCAT_VERSION} ${TOMCAT_HOME} && \
rm -rf ${TOMCAT_HOME}/webapps/* && \
apk del curl && \
```

```
rm -rf /tmp/* /var/cache/apk/*
# 复制配置文件, 具体内容见 Github 仓库代码
COPY logging.properties ${TOMCAT_HOME}/conf/logging.properties
COPY server.xml ${TOMCAT_HOME}/conf/server.xml
```

```
VOLUME ["/logs"]
EXPOSE 8080
```

构建并测试, 然后运行容器。

```
$ docker run -it --rm davidcaste/alpine-tomcat /opt/tomcat/bin/catalina.sh
run
```

复制.war 文件到容器里, 完成部署。

```
$ docker cp ./sample.war tomcat-ci:/opt/tomcat/webapps/sample.war
```

一般情况下, 也把/logs 作为数据卷挂载, 里面有日志文件; Tomcat 的配置目录最好也选择挂载的方式保存。

13.4 其他 Web 服务器

除了上面介绍的 3 种 Web 服务器之外, 还有很多优秀的 Web 服务器被广泛使用。受限于篇幅, 本书不再一一详述。下面选择两款比较有代表性的 Web 服务器简单介绍, 体验在 Docker 环境下运行 Web 服务的便捷与快速。

13.4.1 Caddy 服务器

Caddy (<https://caddyserver.com>) 是一个年轻的 Web 服务器, 默认提供了 HTTP/2 和自动化的 HTTPS 配置。使用 Caddy 可以大大简化前端负载的配置复杂度。而且 Caddy 基于 Golang 开发, 在性能上比较出色, 支持自定义扩展, 配置简单、程序轻量。

Caddy 在 Library 仓库并没有镜像。但是 Caddy Wiki 推荐了几个镜像, 如 abiosoft/caddy、darron/caddy、joshix/caddy、jumanjiman/caddy、zenithar/nano-caddy 等。

本节以 abiosoft/caddy 为例, 演示使用 Docker 运行 Caddy 的方法。与上面的 Web 服务器不同, 运行 Caddy 只在弹指一挥间:

```
$ docker run -d -p 2015:2015 abiosoft/caddy
```

现在打开浏览器, 输入 <http://127.0.0.1:2015> 就可以看到 Caddy 的欢迎界面了。

前面说过, Caddy 是一个自动配置 HTTPS 的 Web 服务器, 它会自动使用 Letsencrypt 提供的 SSL 证书, 到期时自动续期。如果像上面这样启动容器, 意味着每次启动都要从 Letsencrypt 服务器申请一个证书, 这样很容易被 Letsencrypt 拉入黑名单, 因此强烈建议使用数据卷把证书保存到本地。例如:

```
$ docker run -d \
  -v $(pwd)/Caddyfile:/etc/Caddyfile \ # Caddy 的配置文件 (类似 nginx.conf)
  -v $HOME/.caddy:/root/.caddy \      # 保存证书 (证书位置在容器内部的 /root/.
caddy 目录)
  -p 80:80 -p 443:443 \
  abiosoft/caddy
```


当然也可以设置证书保存目录。

```
$ docker run -d \  
-e "CADDYPATH=/etc/caddycerts" \ # 设置 Caddy 的相关目录，包括证书目录  
-v $HOME/.caddy:/etc/caddycerts \  
-p 80:80 -p 443:443 \  
abiosoft/caddy
```

为了方便使用，Caddy 集成了一系列非常方便的工具，如图 13.5 所示。

1. Select Features

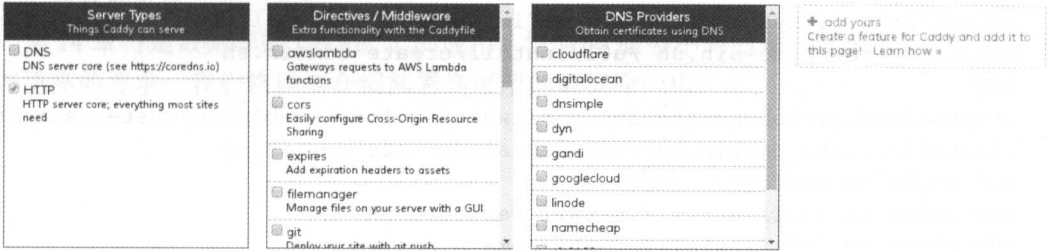


图 13.5 Caddy 扩展

在 abiosoft/caddy 镜像中还有一个标签集成了 PHP 环境如下。

```
$ docker run -d -v /path/to/php/src:/srv -p 2015:2015 abiosoft/caddy:php
```

使用上面这条命令可以立刻把本地的 PHP 项目运行起来，打开 <http://localhost:2015> 即可访问，非常方便。

关于 Caddy 配置文件的语法，可以阅读官方文档 <https://caddyserver.com/docs>。

13.4.2 WebLogic 服务器

WebLogic 是美国 Oracle 公司出品的一个应用服务器。确切的说它是一个基于 Java EE 架构的中间件。WebLogic 是用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 Java 应用服务器。WebLogic 将 Java 的动态功能和 Java Enterprise 标准的安全性引入大型网络应用的开发、集成、部署和管理之中。

本节以 Oracle Linux 7.2、Weblogic 12.2.1、JDK 8u92 为例，讲解 WebLogic 在 Docker 上的应用。Oracle 的 Docker 镜像现在是由 Oracle 维护。

代码参考地址 <https://github.com/iwanttobefreak/docker-weblogic1221>。

```
FROM oraclelinux:7.2  
# 设置变量，构建时使用 --build-arg 参数可以指定这两个值  
# Oracle 安装软件需要账号  
ARG ORACLE_USER  
ARG ORACLE_PASSWORD  
  
# 添加用户  
RUN groupadd -g 1001 weblogic && useradd -u 1001 -g weblogic weblogic  
RUN mkdir -p /u01/install && mkdir -p /u01/scripts  
  
# 解压工具  
RUN yum install -y tar && yum install -y unzip
```



```

COPY scrics/download_weblogic1221.sh /u01/install/download_weblogic1221.sh
COPY scrics/install_weblogic.sh /u01/install/install_weblogic.sh
COPY scrics/oraInst.loc /u01/install/oraInst.loc
COPY scrics/response_file /u01/install/response_file

COPY scrics/create_domain.ini /u01/install/create_domain.ini
COPY scrics/start_AdminServer.sh /u01/scripts/start_AdminServer.sh
COPY scrics/start_nodemanager.sh /u01/scripts/start_nodemanager.sh
COPY scrics/start_ALL.sh /u01/scripts/start_ALL.sh
ADD
https://raw.githubusercontent.com/iwanttobefreak/weblogic/master/scrics
/install/create_domain.sh /u01/install/create_domain.sh
ADD
https://raw.githubusercontent.com/iwanttobefreak/weblogic/master/scrics
/install/create_domain.py /u01/install/create_domain.py
RUN chown -R weblogic. /u01
RUN chmod +x /u01/install/install_weblogic.sh
RUN chmod +x /u01/install/create_domain.sh
RUN chmod +x /u01/scripts/start_nodemanager.sh
RUN chmod +x /u01/scripts/start_AdminServer.sh
RUN chmod +x /u01/scripts/start_ALL.sh

# 指定用户
USER weblogic

ENV USER_MEM_ARGS="-Djava.security.egd=file:/dev/./urandom"

RUN cd /u01/install && \
/u01/install/download_weblogic1221.sh $ORACLE_USER $ORACLE_PASSWORD && \
unzip fmw_12.2.1.1.0_wls_Disk1_1of1.zip

RUN cd /u01/install && /u01/install/install_weblogic.sh

RUN cd /u01/install && /u01/scripts/start_AdminServer.sh && \
./create_domain.sh create_domain.ini

# 删除安装目录
RUN rm -f /u01/install/*

CMD ["/u01/scripts/start_ALL.sh"]

```

执行构建时使用参数:

```

$ docker build --build-arg ORACLE_USER=<oracle username> \
--build-arg ORACLE_PASSWORD=<oracle password> \
-t username/weblogic1221 .

```

构建成功之后使用 **run** 命令运行:

```

$ docker run -d -ti -p 7001:7001 username/weblogic1221

```

现在可以使用浏览器打开 <http://localhost:7001/console> 访问控制台了。用户名为 **weblogic**; 密码为 **weblogic01**。

13.5 本章小结

本章详细介绍了 Apache、Nginx 和 Tomcat 这 3 个 Web 服务器在 Docker 上的应用，并尝试基于 Debian 和 Alpine 构建它们的镜像。最后还选择了两个比较有代表性的 Web 服务器 Caddy 和 WebLogic 进行简单介绍。这两个镜像虽然没有在 Library 仓库中维护，但是凭借它们活跃的社区，不可不谈。

第 14 章将是数据库在 Docker 上的应用。与 Web 服务不同的是，数据库会在存储方面有着更高的要求，我们将详细介绍部署高可用的数据库应用。

第 14 章 数 据 库

数据库 (Database) 是按照数据结构来组织、存储和管理数据的仓库, 目前主流数据库有 关系数据库 (SQL) 和非关系数据库 (NoSQL) 方案, 关系数据库 (Relational database) 是创建在关系模型基础上的数据库, 借助于集合代数等数学概念和方法来处理数据库中的数据。目前主流的关系数据库有 Oracle、SQLserver、MySQL、PostgreSQL 等。

非关系型数据库是对不同于传统的关系数据库的数据库管理系统的统称。两者存在许多显著的不同点, 其中最重要的是非关系型数据库不使用 SQL 作为查询语言, 其数据存储可以不需要固定的表格模式, 也经常避免使用 SQL 的 JOIN 操作, 一般有水平可扩展性的特征。

本章将选取比较有代表性的几款数据库软件, 通过在 Docker 中构建部署使用它们来展示 Docker 在数据应用方面的能力, 本章对读者要求不高, 只需要对数据库的基本操作有一定了解即可。

14.1 MySQL 数据库

MySQL 的大名相信读者已经如雷贯耳, MySQL 性能高、成本低、可靠性好, 已经成为最流行的开源数据库, 被广泛地应用在互联网上的各种网站中。非常流行的开源软件组合 LAMP/LNMP 中的 M 指的就是 MySQL。

14.1.1 官方镜像的剖析与使用

使用 pull 操作可以拉取相应的 MySQL 版本到本地:

```
$ docker pull mysql
```

从 https://hub.docker.com/r/_/mysql/tags/ 上可以查看 MySQL 的标签, 如图 14.1 所示, 以确定想要拉取的版本。默认是 latest 版本, 也就是 5.7 的 MySQL。

- 5.7.15, 5.7, 5, latest (5.7/Dockerfile)
- 5.6.33, 5.6 (5.6/Dockerfile)
- 5.5.52, 5.5 (5.5/Dockerfile)

图 14.1 MySQL 版本与对应的 Dockerfile 链接

为了了解官方镜像的内容, 从 Github 上可以查看相应的 Dockerfile, 比如 MySQL 5.6 的 Dockerfile。

```

FROM debian:jessie
.....
# 这一步是为了添加一个 MySQL 用户组与 MySQL 用户, 以便在后面以非 root 状态运行
MySQL, 提高安全性
RUN groupadd -r mysql && useradd -r -g mysql mysql
.....
# 统一设置版本号, 修改起来更便捷
ENV MYSQL_MAJOR 5.6
ENV MYSQL_VERSION 5.6.33-1debian8
# 添加软件源
RUN echo "deb http://repo.mysql.com/apt/debian/ jessie mysql-${MYSQL_
MAJOR}" > /etc/apt/sources.list.d/mysql.list
.....
# 设置初始 MySQL 环境并安装
<省略部分> apt-get install -y mysql-server="${MYSQL_VERSION}" <省略部分>
.....
# 设置数据卷, 方便备份恢复
VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards
compat
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld"]

```

通过上面的资料, 已经大体知道了这个镜像的结构, 接下来使用这个镜像来运行一个 MySQL 容器。

```

$ docker run --name test-mysql --rm -e MYSQL_ROOT_PASSWORD=my-secret-pw
mysql:5.6
.....
Version: '5.6.32'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306
MySQL Community Server (GPL)

```

在实验过程中, 通常加上--rm 参数以便实验结束时自动删除容器。在上面的例子中, 我们启动了一个 MySQL 容器, 通过-e MYSQL_ROOT_PASSWORD 指定了数据库 root 用户的密码, 因为启动命令中没有设置暴露端口, 所以无法从外部访问到数据库, 当出现数据库版本号时表示数据库已经成功启动。

通过官方镜像自带的 MySQL 客户端可以连接到上面的容器中。

```

$ docker run -it --link test-mysql:mysql --rm mysql:5.6 sh -c \
'exec mysql -h"$MYSQL_PORT_3306_TCP_ADDR" \
-P"$MYSQL_PORT_3306_TCP_PORT" \
-uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'

```

```

Warning: Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.6.32 MySQL Community Server (GPL)

```

```

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

```

```

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

```

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| performance_schema |
| registry   |
+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

这里的`--link`在前面的章节中都介绍过,通过该参数可以连接容器,使容器之间可以通信。有了这个例子,不妨再扩展一下,使用`--link`来连接一个需要使用MySQL的容器:

```
$ docker run --name my-app --link test-mysql:mysql -d my-application-can-link-mysql
```

这样数据库容器并不需要暴露3306端口,不仅确保了数据库安全,还方便了管理人员对端口的调整。

如果有需要进入数据库容器可以使用:

```
$ docker exec -it test-mysql bash
```

而查看数据库的日志,可以使用:

```
$ docker logs test-mysql
```

上面只是对数据库的基本应用,在很多时候需要手动修改配置,以适应用户的需求,还需要对数据库进行备份等操作。

官方提供的MySQL镜像的配置文件默认在`/etc/mysql/conf.d`位置,所以使用`-v`参数可以把自己的配置放进容器里面。

```
$ docker run --name some-mysql \
-v /my/custom:/etc/mysql/conf.d \
-e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

`/my/custom`就是用户自己的配置文件目录。

除了自定义的配置,还需要有备份的功能,在上面Dockerfile中已经提到过,数据卷的目录是`/var/lib/mysql`,所以在启动时只需要设置参数`-v`即可。

```
$ docker run --name some-mysql \
-v /my/custom:/etc/mysql/conf.d \
-v /my/data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

14.1.2 构建自己的MySQL镜像

构建自己的数据库镜像一般来说都从自己的需求开始定制,所以在构建前先写好配置文件。

`mysqld_charset.cnf` (设置数据库字符集) 文件如下:

```
[mysqld]
character_set_server=utf8
character_set_filesystem=utf8
```

```
collation-server=utf8_general_ci
init-connect='SET NAMES utf8'
init_connect='SET collation_connection = utf8_general_ci'
skip-character-set-client-handshake
```

my.cnf（基本配置）文件如下：

```
[mysqld]
bind-address=0.0.0.0
skip_name_resolve
#server-id
#log-bin
```

这时就可以写 Dockerfile 了，首先选择一个基础镜像，这里用 ubuntu:trusty 作为基础镜像。

```
FROM ubuntu:trusty
```

添加上面写好的配置：

```
COPY my.cnf /etc/mysql/conf.d/my.cnf
COPY mysqld_charset.cnf /etc/mysql/conf.d/mysqld_charset.cnf
```

更新本地软件源缓存，然后安装 mysql-server，安装 pwgen 可以自动生成密码。

安装软件并清理

```
RUN apt-get update && \
    apt-get -yq install mysql-server-5.6 pwgen && \
    rm -rf /var/lib/apt/lists/* && \
    rm /etc/mysql/conf.d/mysqld_safe_syslog.cnf && \
    if [! -f /usr/share/mysql/my-default.cnf] ; then cp /etc/mysql/my.cnf \
    /usr/share/mysql/my-default.cnf; fi && \
    mysql_install_db > /dev/null 2>&1 && \
    touch /var/lib/mysql/.EMPTY_DB
```

添加 MySQL 脚本（参考 Tutum 公司（现已被 Docker 收购）的代码，Github 仓库地址为 <https://github.com/tutumcloud/mysql/tree/master/5.6>）如下：

```
COPY import_sql.sh /import_sql.sh
COPY run.sh /run.sh
```

设置 MySQL 的常用变量，这些变量意味着可以在启动时使用 -e 来设置变量的具体内容：

```
ENV MYSQL_USER=admin \
    MYSQL_PASS=**Random** \
    ON_CREATE_DB=**False** \
    REPLICATION_MASTER=**False** \
    REPLICATION_SLAVE=**False** \
    REPLICATION_USER=replica \
    REPLICATION_PASS=replica
```

设置数据卷如下：

```
# Add VOLUMES to allow backup of config and databases
VOLUME ["/etc/mysql", "/var/lib/mysql"]
```

最后别忘了端口与启动命令：

```
EXPOSE 3306
CMD ["/run.sh"]
```

这样就完成了一份 MySQL 的 Dockerfile，构建之后运行时会自动生成一个 admin 密码。

```
=====
You can now connect to this MySQL Server using:
```

```
mysql -uadmin -p47nnf4FweaKu -h<host> -P<port>
```

Please remember to change the above password as soon as possible!
MySQL user 'root' has no password but only allows local connections.

使用 `mysql -uadmin -p` 可以连接到数据库，此时为确保安全，不能使用 `root` 用户在外
部连接到容器。与官方镜像一样，数据文件都保存在 `/var/lib/mysql` 目录下。

在这个镜像中，可以在启动时设置主从数据库，Master 数据库如下：

```
$ docker run -d \
  -e REPLICATION_MASTER=true \
  -e REPLICATION_PASS=mypass \
  -p 3306:3306 \
  --name mysql yourname/mysql
```

Slave 数据库如下：

```
$ docker run -d \
  -e REPLICATION_SLAVE=true \
  -p 3307:3306 \
  --link mysql:mysql yourname/mysql
```

现在可以通过 3306 和 3307 分别连接主从数据库了。需要注意的是，使用 MySQL 5.6
构建的镜像，它的容器数据卷不能被其他版本的 MySQL 容器读取。

14.2 PostgreSQL 数据库

PostgreSQL 是自由的对象关系型数据库服务器（数据库管理系统），在灵活的 BSD
许可证下发行。它在其他开放源代码数据库系统（如 MySQL 和 Firebird），以及专有系统
如 Oracle、Sybase、IBM 的 DB2 和 Microsoft SQL Server 之外，为用户提供了又一种选择。

14.2.1 官方镜像的使用

1. 基础应用

从 Docker Hub 上拉取 PostgreSQL 的命令是：

```
$ docker pull postgres
```

目前 PostgreSQL 在 Docker Hub 上有如图 14.2 所示的标签，本节讲解以 9.6 版本为主。



- 9.6.0, 9.6, 9, latest (9.6/Dockerfile)
- 9.5.4, 9.5 (9.5/Dockerfile)
- 9.4.9, 9.4 (9.4/Dockerfile)
- 9.3.14, 9.3 (9.3/Dockerfile)
- 9.2.18, 9.2 (9.2/Dockerfile)
- 9.1.23, 9.1 (9.1/Dockerfile)

图 14.2 Docker Hub 上的 PostgreSQL 版本

启动一个 postgres 实例并不复杂，与 MySQL 类似需要指定一个 POSTGRES_PASSWORD 变量：

```
$ docker run --name some-postgres \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -d postgres
```

这个镜像的 EXPOSE 指令的值是 5432，也就是说可以使用 -p 参数来映射，当然如果是容器通信，直接使用--link 是最好的办法。

例如，连接一个需要数据库的应用：

```
$ docker run --name some-app \
  --link some-postgres:postgres \
  -d application-that-uses-postgres
```

在这个应用中，数据库的地址并非 localhost，而是 postgres，前面提到--link 会把容器名写入/etc/hosts 文件，相当于 hostname。

与 MySQL 一样，这个镜像同样可以作为客户端连接到已有数据库：

```
$ docker run -it --rm --link some-postgres:postgres postgres psql -h postgres
-U postgres
Password for user postgres:
psql (9.6.0)
Type "help" for help.
```

```
postgres=#
You are using psql, the command-line interface to PostgreSQL.
Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit
postgres=# \l
```

List of databases					
Name	Owner	Encoding	Collate	Ctype	Access privileges
postgres	postgres	UTF8	en_US.utf8	en_US.utf8	
template0	postgres	UTF8	en_US.utf8	en_US.utf8	=c/postgres +
postgres=CTc/postgres					
templatel	postgres	UTF8	en_US.utf8	en_US.utf8	=c/postgres +
postgres=CTc/postgres					
(3 rows)					

```
postgres=#
```

PostgreSQL 镜像有以下几个变量可以在启动时设定。

- **POSTGRES_PASSWORD**: 推荐在使用 PostgreSQL 镜像时设定该变量，这个变量的作用是设置超级管理员的密码。超级管理员的用户名可以使用 POSTGRES_USER 变量设定，如上面的例子，超级管理员的密码是 mysecretpassword。
- **POSTGRES_USER**: 该变量通常是结合上面的 POSTGRES_PASSWORD 变量来设置超级管理员的账号和密码。该变量会创建一个拥有超级管理员权限的用户和一个同名的数据库。如果不指定会使用默认的 postgres 用户名。
- **PGDATA**: 这个可选的环境变量可以指定一个位置用于存储数据库文件。默认位置

是/var/lib/postgresql/data。

- **POSTGRES_DB**: 这个可选的环境变量可以指定默认数据库的名称。就是镜像启动时创建的数据库, 如果不指定, 会默认使用 **POSTGRES_USER** 的值, 如果 **POSTGRES_USER** 没设置则使用 **postgres** 这个名字。
- **POSTGRES_INITDB_ARGS**: 这个可选的环境变量可以发送参数给初始化数据库的命令 **postgres initdb**, 参数的值使用空格分开, 如校验数据页面 **-c POSTGRES_INITDB_ARGS="--data-checksums"**。

2. 扩展应用

PostgreSQL 的镜像提供了一个扩展的功能, 如果想在镜像启动时执行一些额外的动作就需要用到这个功能了。在 PostgreSQL 镜像中有一个扩展目录位于 **/docker-entrypoint-initdb.d** 中, 可以存放用户的初始化脚本 (.sh) 或者数据库文件 (.sql), 在镜像执行完成 **initdb** 的任务之后, 会执行这个目录下的脚本。

例如添加一个用户和数据库, 挂载一个数据卷在 **/docker-entrypoint-initdb.d** 目录下, 预先放置 **init-user-db.sh** (也可以从 Dockerfile 中添加, 然后重新构建)。

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
CREATE USER docker;
CREATE DATABASE docker;
GRANT ALL PRIVILEGES ON DATABASE docker TO docker;
EOSQL
```

启动之后会默认执行该脚本, 在这个动作中, 脚本默认编码是 **en_US.utf8**, SQL 文件运行的用户是 **POSTGRES_USER** 指定的值。

用户可以指定数据库的默认编码 (需要重新构建):

```
FROM postgres:9.4
RUN localedef -i de_DE -c -f UTF-8 -A /usr/share/locale/locale.alias
de_DE.UTF-8
ENV LANG de_DE.utf8
```

最后提醒一点, 要保存容器的数据文件不要忘记挂载 **/var/lib/postgresql/data** 为数据卷。

14.2.2 官方镜像的剖析

PostgreSQL 的 Dockerfile 并不复杂, 在 Github 目录中也仅两个文件, 其中 Dockerfile 文件如下:

```
# 基于 Debian 构建
FROM debian:jessie

# 创建并设定容器执行命令的用户
RUN groupadd -r postgres --gid=999 && useradd -r -g postgres --uid=999
postgres

# 安装 gosu
ENV GOSU_VERSION 1.7
RUN set -x \
```

```

    && apt-get update && apt-get install -y --no-install-recommends
ca-certificates wget && rm -rf /var/lib/apt/lists/* \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/
releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tianon/gosu/
releases/download/$GOSU_VERSION/gosu-$(dpkg --print-architecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007
F00F88E364FD4036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget

```

设置 en_US.UTF-8 为默认编码

```

RUN apt-get update && apt-get install -y locales && rm -rf /var/
lib/apt/lists/* \
    && localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias
en_US.UTF-8
ENV LANG en_US.utf8

```

创建这个文件夹用于执行用户自定义的初始化动作

```
RUN mkdir /docker-entrypoint-initdb.d
```

下面开始安装 PostgreSQL

```
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
```

```
ENV PG_MAJOR 9.6
```

```
ENV PG_VERSION 9.6.0-1.pgdg80+1
```

```

RUN echo 'deb http://apt.postgresql.org/pub/repos/apt/ jessie-pgdg main'
$PG_MAJOR > /etc/apt/sources.list.d/pgdg.list

```

```

RUN apt-get update \
    && apt-get install -y postgresql-common \
    && sed -ri 's/#(create_main_cluster) .*$/\1 = false/' /etc/postgresql-
common/createcluster.conf \
    && apt-get install -y \
        postgresql-$PG_MAJOR=$PG_VERSION \
        postgresql-contrib-$PG_MAJOR=$PG_VERSION \
    && rm -rf /var/lib/apt/lists/*

```

示例配置

```

RUN mv -v /usr/share/postgresql/$PG_MAJOR/postgresql.conf.sample /usr/
share/postgresql/ \
    && ln -sv ../postgresql.conf.sample /usr/share/postgresql/$PG_MAJOR/ \
    && sed -ri "s!^#?(listen_addresses)\s*=\s*\S+.*!\1 = '*'! " /usr/
share/postgresql/postgresql.conf.sample

```

```
RUN mkdir -p /var/run/postgresql && chown -R postgres /var/run/postgresql
```

设置基本 PostgreSQL 变量

```
ENV PATH /usr/lib/postgresql/$PG_MAJOR/bin:$PATH
```

设置数据文件存储位置与数据卷信息

```

ENV PGDATA /var/lib/postgresql/data
VOLUME /var/lib/postgresql/data

```

```
# 复制接入点脚本，用于执行数据库初始化动作
COPY docker-entrypoint.sh /

ENTRYPOINT ["/docker-entrypoint.sh"]

# 设置端口暴露
EXPOSE 5432
CMD ["postgres"]
```

从上面的 Dockerfile 文件中可以看出，初始化脚本是作为 ENTRYPOINT 写入镜像中的，这意味着如果删除默认创建的数据库可能会导致外部编排工具无法连接到这个容器中。当然，删除容器之后重新启动创建也不失为一个补救办法。

14.2.3 构建 PostgreSQL 镜像

官方镜像使用的是 Debian 构建的数据库镜像，镜像体积有 100 多 MB，实际上镜像的体积可以大大减小，本节使用 Alpine 来构建一个更小的 PostgreSQL 镜像。

首先要明确 Alpine 是否有 PostgreSQL 的软件包，在 Alpine 软件列表 (http://nl.alpinelinux.org/alpine/edge/main/x86_64/)中可以看到有该软件包，如图 14.3 所示。

postfix-mysql-3.1.3-r0.apk	2016-Oct-10 15:52:46	7.0K	application/x-tgz
postfix-pcre-3.1.3-r0.apk	2016-Oct-10 15:52:46	6.7K	application/x-tgz
postfix-pgsql-3.1.3-r0.apk	2016-Oct-10 15:52:46	6.8K	application/x-tgz
postfix-policyd-spf-perl-2.007-r2.apk	2016-Jul-27 17:25:43	11.4K	application/x-tgz
postfix-sqlite-3.1.3-r0.apk	2016-Oct-10 15:52:46	5.0K	application/x-tgz
postfix-stone-3.1.3-r0.apk	2016-Oct-10 15:52:46	26.7K	application/x-tgz
postfixadmin-2.93-r1.apk	2016-Aug-13 11:35:23	1.2M	application/x-tgz
postgresql-9.6.0-r1.apk	2016-Oct-10 15:52:46	5.0M	application/x-tgz
postgresql-bdr-9.4.9-pl1-r0.apk	2016-Oct-10 15:52:46	5.1M	application/x-tgz
postgresql-bdr-contrib-9.4.9-pl1-r0.apk	2016-Oct-10 15:52:46	552.3K	application/x-tgz
postgresql-bdr-dev-9.4.9-pl1-r0.apk	2016-Oct-10 15:52:46	1.0M	application/x-tgz
postgresql-bdr-extension-0.9.3-r0.apk	2016-Jan-22 15:17:45	1.2M	application/x-tgz
postgresql-bdr-extension-doc-0.9.3-r0.apk	2016-Jan-22 15:17:45	6.9K	application/x-tgz
postgresql-client-9.6.0-r1.apk	2016-Oct-10 15:52:46	170.3K	application/x-tgz
postgresql-contrib-9.6.0-r1.apk	2016-Oct-10 15:52:46	515.1K	application/x-tgz
postgresql-dev-9.6.0-r1.apk	2016-Oct-10 15:52:46	1.3M	application/x-tgz
postgresql-doc-9.6.0-r1.apk	2016-Oct-10 15:52:47	3.8M	application/x-tgz
postgresql-libs-9.6.0-r1.apk	2016-Oct-10 15:52:47	63.7K	application/x-tgz
postgrey-1.36-r2.apk	2016-Jun-07 07:44:13	60.9K	application/x-tgz
pound-2.7-r2.apk	2016-Oct-10 15:52:47	45.9K	application/x-tgz
pound-doc-2.7-r2.apk	2016-Oct-10 15:52:47	13.8K	application/x-tgz
powertop-2.8-r0.apk	2015-Dec-09 20:00:26	241.6K	application/x-tgz
powertop-doc-2.8-r0.apk	2015-Dec-09 20:00:26	6.8K	application/x-tgz

图 14.3 Alpine 软件包

因此可以直接从软件仓库安装：

```
FROM alpine:edge
```

首先使用 Alpine 作为基础镜像，然后添加软件仓库列表如下：

```
RUN echo "@edge http://nl.alpinelinux.org/alpine/edge/main" >> /etc/apk/repositories
```

接下来是更新本地软件列表，并安装 PostgreSQL 数据库，建议也安装 postgresql-contrib 软件包。

```
RUN apk update && \
    apk add curl "postgresql@edge<9.6" "postgresql-contrib@edge<9.6"
```

然后像官方镜像那样，需要提供一个用户自定义的脚本目录。

```
RUN mkdir /docker-entrypoint-initdb.d
```

安装 gosu 并清理缓存。

```
RUN curl -o /usr/local/bin/gosu \
    -sSL
```

```
"https://github.com/tianon/gosu/releases/download/1.2/gosu-amd64"
RUN chmod +x /usr/local/bin/gosu && \
    apk del curl && \
    rm -rf /var/cache/apk/*
```

设定数据库的默认编码，与官方镜像一样即可。

```
ENV LANG en_US.utf8
ENV PGDATA /var/lib/postgresql/data
VOLUME /var/lib/postgresql/data
```

下面的脚本需要做些修改，主要是 Alpine 中没有 bash 环境，需要替换一些操作。

```
COPY docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

暴露端口和 CMD 指令保持与官方一致，这样在使用方法上比较贴近官方镜像。

```
EXPOSE 5432
CMD ["postgres"]
```

完成这些工作之后，可以整合一下整个 Dockerfile 和 docker-entrypoint.sh 如下：

```
FROM alpine:edge

RUN echo "@edge http://nl.alpinelinux.org/alpine/edge/main" >> /etc/apk/
repositories && \
    apk update && \
    apk add curl "postgresql@edge<9.6" "postgresql-contrib@edge<9.6" && \
    mkdir /docker-entrypoint-initdb.d && \
    curl -o /usr/local/bin/gosu \
        -sSL "https://github.com/tianon/gosu/releases/download/1.2/gosu-amd64"
&& \
    chmod +x /usr/local/bin/gosu && \
    apk del curl && \
    rm -rf /var/cache/apk/*

ENV LANG en_US.utf8
ENV PGDATA /var/lib/postgresql/data
VOLUME /var/lib/postgresql/data

COPY docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]

EXPOSE 5432
CMD ["postgres"]
```

然后将镜像的 RUN 指令合并为一条，docker-entrypoint.sh 的内容如下：

```
#!/bin/sh
# 因为没有建立 postgres 用户，所以要使用 chown 改变文件夹所有权。
chown -R postgres "$PGDATA"

# 初始化数据库
if [ -z "$(ls -A "$PGDATA")" ]; then
    gosu postgres initdb
    sed -ri "s/^(listen_addresses\s*=\s*)\S+/\1'/'/" "$PGDATA"/postgresql.conf

    : ${POSTGRES_USER:="postgres"}
    : ${POSTGRES_DB:=$POSTGRES_USER}

    if [ "$POSTGRES_PASSWORD" ]; then
        pass="PASSWORD '$POSTGRES_PASSWORD'"
        authMethod=md5
    else

```

```

    echo "=====
    echo "!!! Use \${POSTGRES_PASSWORD} env var to secure your database !!!"
    echo "=====
    pass=
    authMethod=trust
fi
echo

if [ "${POSTGRES_DB}" != 'postgres' ]; then
    createSql="CREATE DATABASE ${POSTGRES_DB};"
    echo $createSql | gosu postgres postgres --single -jE
    echo
fi

if [ "${POSTGRES_USER}" != 'postgres' ]; then
    op=CREATE
else
    op=ALTER
fi

userSql="$op USER ${POSTGRES_USER} WITH SUPERUSER $pass;"
echo $userSql | gosu postgres postgres --single -jE
echo

# internal start of server in order to allow set-up using psql-client
# does not listen on TCP/IP and waits until start finishes
gosu postgres pg_ctl -D "$PGDATA" \
    -o "-c listen_addresses=''" \
    -w start

echo
for f in /docker-entrypoint-initdb.d/*; do
    case "$f" in
        *.sh) echo "$0: running $f"; . "$f" ;;
        *.sql) echo "$0: running $f"; psql --username "${POSTGRES_USER}"
--dbname "${POSTGRES_DB}" < "$f" && echo ;;
        *) echo "$0: ignoring $f" ;;
    esac
    echo
done

gosu postgres pg_ctl -D "$PGDATA" -m fast -w stop

{ echo; echo "host all all 0.0.0.0/0 $authMethod"; } >> "$PGDATA"/
pg_hba.conf
fi

exec gosu postgres "$@"

```

上面的完整代码可以参考 <https://github.com/kiasaki/docker-alpine-postgres>。

PostgreSQL 镜像按照官方的 Dockerfile 定义来创建，因此用法与官方镜像一致，但是镜像体积大大减小（仅 16 MB）了。

14.2.4 数据备份与导入

备份 PostgreSQL 首先想到的就是在 PostgreSQL 实例上运行 `pg_dump`，最简单的办法就是使用 `docker exec` 命令：

```
$ docker exec postgres pg_dump -h db -f /volume/path/backup.sql
```

上面的/volume/path/表示的是数据卷路径。

如果镜像中已经包含了 `postgresql-client` 软件包，直接使用该工具即可。

```
$ docker run -it --link postgres:db postgres /usr/bin/pg_dump -h db
```

如果镜像中没有该工具，可以手动构建一个镜像，`Dockerfile` 文件只有以下几行。

```
FROM debian:wheezy
RUN apt-get update -y && \
    apt-get install -y postgresql-client && \
    apt-get clean -y
ENTRYPOINT ["/usr/bin/pg_dump"]
```

然后使用 `docker build -t my/pg_dump` 执行构建，就可以备份数据库了。

导入数据就更简单了，直接使用 PostgreSQL 的导入命令即可。

14.3 Redis 数据库

REmote DIctionary Server (Redis)是一个由 Salvatore Sanfilippo 写的 key-value 存储系统。

Redis 是一个开源的使用 ANSI C 语言编写，遵守 BSD 协议，支持网络，可基于内存亦可持久化的日志型 Key-Value 数据库，并提供多种语言的 API。它通常被称为数据结构服务器，因为值(value)可以是字符串(String)，哈希(Map)，列表(list)，集合(sets)和有序集合(sorted sets)等类型。

14.3.1 官方镜像的使用

目前 Redis 在 Docker Hub 上共有如图 14.4 所示的几个标签，其中包括基于 Debian 和 Alpine 构建的镜像，除此之外，还可以看到一个特别的 32bit 标签，这里的 32bit 并非是指这个镜像是 32 位的，也不表示这个镜像的实例可以运行在 32 位机器上，而是指这个镜像里面包含了 32 位构建的二进制文件。



redis

- 3.0.7, 3.0 (3.0/Dockerfile)
- 3.0.7-32bit, 3.0-32bit (3.0/32bit/Dockerfile)
- 3.0.7-alpine, 3.0-alpine (3.0/alpine/Dockerfile)
- 3.2.4, 3.2, 3, latest (3.2/Dockerfile)
- 3.2.4-32bit, 3.2-32bit, 3-32bit, 32bit (3.2/32bit/Dockerfile)
- 3.2.4-alpine, 3.2-alpine, 3-alpine, alpine (3.2/alpine/Dockerfile)

图 14.4 Redis 镜像

从 Docker Hub 拉取 Redis 镜像：

```
$ docker pull redis
```

Docker 会输出 Redis 镜像的信息、下载状态，下载完成之后系统会显示最终状态信息。镜像拉取完成之后，大家可以用下面的命令启动 Redis 容器：

```
$ docker run --name some-redis -d redis
```


-d 参数的作用是让 Redis 在后台运行，因为本例中采用这种后台运行的方式较为合适，所以这里写上了这个参数。

在 Dockerfile 中，EXPOSE 指令的值是 6379，因此如果需要映射宿主机端口，可以使用 -p <port>:6379，一般更推荐使用 --link 参数使用数据库。

启动为永久存储容器，可以使用 Redis 的命令：

```
$ docker run --name some-redis -d redis redis-server --appendonly yes
```

如果启用了持久性特性，那么数据会保存在数据卷中，Redis 镜像在 Dockerfile 中定义了 VOLUME 指令，指定 /data 目录为数据卷，因此可以使用 --volumes-from some-volume-container 或者 -v /docker/host/dir:/data 来挂载数据卷。

一般来说使用 Redis 时都是需要一个连接到 Redis 的应用。

```
$ docker run --name my_app --link some-redis:redis -d my_application
```

如果需要使用 Redis 的来存储操作数据，可以使用 redis-cli 来连接。

```
$ docker run -it --link some-redis:redis --rm redis redis-cli -h redis -p 6379
```

接下来就可以通过 set 和 put 命令来执行 Redis 的存取操作了，使用 --rm 参数可以在退出容器之后删除容器。

另外，如果需要使用自己的 redis.conf 配置文件，那么可以使用 Dockerfile 基于当前的 Redis 镜像构建一个自己的镜像，例如：

```
FROM redis
COPY redis.conf /usr/local/etc/redis/redis.conf
CMD [ "redis-server", "/usr/local/etc/redis/redis.conf" ]
```

当然其实更好的办法就是使用数据卷，把本地的配置文件挂载到容器中：

```
$ docker run -v /path/conf/redis.conf:/usr/local/etc/redis/redis.conf \
  --name myredis \
  redis redis-server /usr/local/etc/redis/redis.conf
```

配置文件存放在 /usr/local/etc/redis/ 目录中。

14.3.2 构建自己的 Redis 镜像

要构建一个 Redis 镜像，首先要清楚需要什么依赖与工具，通常情况下，按照 Redis 官网的安装方式去操作即可。

首先需要一份启动脚本，这里参考的是官方的启动脚本。

```
#!/bin/sh
set -e

# 第一个参数可以是 '-f' 或者其他参数 '--some-option'
# 或者是一个配置文件 'something.conf'
if [ "${1#-}" != "$1" ] || [ "${1%.conf}" != "$1" ]; then
    set -- redis-server "$@"
fi

# 允许使用 '--user' 参数切换指定用户来启动容器
if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then
    chown -R redis .
```

```

    exec su-exec redis "$0" "$@"
fi

exec "$@"

```

脚本并不复杂,实际上执行操作的是 Redis 的命令行工具。下面根据需要调整 Dockerfile 内容如下:

```

FROM alpine:3.4

ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.0.7.tar.gz
ENV REDIS_DOWNLOAD_SHA1 e56b4b7e033ae8dbf311f9191cf6fdf3ae974d1c

COPY docker-entrypoint.sh /usr/local/bin/

# 添加容器程序命令需要的用户与用户组
RUN addgroup -S redis && adduser -S -G redis redis \

# 安装 su-exec
&& apk add --no-cache 'su-exec>=0.2' \
&& set -x \

# 安装构建依赖
&& apk add --no-cache --virtual .build-deps \
    gcc \
    linux-headers \
    make \
    musl-dev \
    tar \

# 安装 Redis
&& wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
&& echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c - \
&& mkdir -p /usr/src/redis \
&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
&& rm redis.tar.gz \
&& make -C /usr/src/redis \
&& make -C /usr/src/redis install \
&& rm -r /usr/src/redis \

# 移除安装依赖
&& apk del .build-deps \

# 创建数据卷目录,设置启动脚本的执行权限
&& mkdir /data && chown redis:redis /data \
&& chmod +x /usr/local/bin/ docker-entrypoint.sh

WORKDIR /data
VOLUME /data
EXPOSE 6379

ENTRYPOINT ["docker-entrypoint.sh"]
CMD [ "redis-server" ]

```

最后使用 `docker build` 命令构建镜像即可。镜像的使用方法与官方镜像完全一致,但在体积与结构上更清晰精简。

14.4 MongoDB 数据库

MongoDB 是一个基于分布式文件存储的数据库,由 C++ 语言编写,旨在为 Web 应用提供可扩展的高性能数据存储解决方案。

MongoDB 是专为可扩展性、高性能和高可用性而设计的数据库,可以从单服务器部署扩展到大型、复杂的多数据中心架构。利用内存计算的优势, MongoDB 能够提供高性能的数据读写操作。MongoDB 的本地复制和自动故障转移功能使用户的应用程序具有企业级的可靠性和操作灵活性。

14.4.1 官方镜像的使用

MongoDB 镜像在 Docker Hub 的 Library 仓库一共有如图 14.5 所示的几个版本,其中 windowsservercore 标签表示该镜像是使用 microsoft/windowsservercore 作为基础镜像构建的,其实大部分数据库都可以运行在 Windows 平台中。

实际上在 Windows 平台中使用基于 Linux 构建的镜像也可以正常运行, MongoDB 镜像的好处在于可以为 Windows 下的软件提供数据库服务。



- 2.6.12, 2.6, 2 (2.6/Dockerfile)
- 3.0.12, 3.0 (3.0/Dockerfile)
- 3.0.12-windowsservercore, 3.0-windowsservercore
(3.0/windows/windowsservercore/Dockerfile)
- 3.2.10, 3.2, 3, latest (3.2/Dockerfile)
- 3.2.10-windowsservercore, 3.2-windowsservercore, 3-windowsservercore,
windowsservercore (3.2/windows/windowsservercore/Dockerfile)
- 3.3.15, 3.3 (3.3/Dockerfile)
- 3.3.15-windowsservercore, 3.3-windowsservercore
(3.3/windows/windowsservercore/Dockerfile)

图 14.5 MongoDB 镜像

启动一个 MongoDB 镜像实例非常简单:

```
$ docker run --name some-mongo -d mongo
```

镜像的 Dockerfile 中的 EXPOSE 暴露了 27017 作为映射端口,因此如果需要映射端口,请加上 -p <port>:27017 参数,容器互联时不需要映射端口, Docker 会自动在容器之间连接这些暴露端口。

如果用户有容器需要连接一个 MongoDB 数据库容器,可以使用 --link 连接。

```
$ docker run --name some-app --link some-mongo:mongo -d application_image
```

这样在应用容器中使用 mongo 作为数据库就可以连接到 MongoDB 数据库容器。

如需要直接连接到容器中的数据库时,可以使用容器提供的如下命令行工具:

```
$ docker run -it --rm \
  --link some-mongo:mongo mongo \
  sh -c 'exec mongo \
    "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/test"'
```

上面实际上是启动了一个临时的 MongoDB 容器，然后使用容器中的 `mongo` 命令行工具连接到已经启动的 MongoDB 数据库容器中。

在 `Dockerfile` 的 `ENTRYPOINT` 指令指定了 `/entrypoint.sh` 脚本作为接入点，而 `CMD` 指令中指定了 `mongod` 作为启动命令，因此可以使用 MongoDB 的命令在启动时配置数据库，例如，使用 `wiredTiger` 作为数据库存储引擎可以直接使用 `--storageEngine` 参数。

```
$ docker run --name some-mongo -d mongo --storageEngine wiredTiger
```

MongoDB 默认不需要身份验证，但是可以配置。详细的配置过程不是本书的讨论范围，可以在官方文档中阅读更多信息，地址为 <https://docs.mongodb.org/manual/core/authentication/>。

本书只做简单的演示，首先新建容器并设置 `--auth` 参数。

```
$ docker run --name some-mongo -d mongo --auth
```

然后使用 `docker exec` 进入容器。

```
$ docker exec -it some-mongo mongo admin
connecting to: admin
```

然后使用 `db.createUser` 创建一个用户和密码。

```
> db.createUser({ user: 'jsmith', pwd: 'some-initial-password', roles:
[ { role: "userAdminAnyDatabase", db: "admin" } ] });
Successfully added user: {
  "user" : "jsmith",
  "roles" : [
    {
      "role" : "userAdminAnyDatabase",
      "db" : "admin"
    }
  ]
}
```

这时从外部连接这个容器：

```
$ docker run -it --rm \
  --link some-mongo:mongo mongo \
  mongo -u jsmith -p some-initial-password \
  --authenticationDatabase admin some-mongo/some-db
> db.getName();
some-db
```

MongoDB 的数据文件保存在 `/data/db` 目录中，因此使用 `-v` 来设置数据卷可以保存容器的数据文件。

```
$ docker run --name some-mongo -v /my/own/datadir:/data/db -d mongo:tag
```

数据库的导入导出等备份工作可以在 MongoDB 官网的文档中找到详细资料，使用 `docker exec` 或者启动一个临时容器连接操作即可。

14.4.2 构建自己的 MongoDB 镜像

构建一个 MongoDB 镜像非常简单，因为软件列表有 MongoDB 镜像软件包，因此直接从软件仓库安装即可。

Dockerfile 文件如下：

```

FROM alpine:edge

COPY run.sh /
RUN echo http://dl-4.alpinelinux.org/alpine/edge/testing >> /etc/apk/
repositories && \
    apk add --no-cache mongodb && \
    chmod +x /run.sh

VOLUME /data/db
EXPOSE 27017

ENTRYPOINT [ "/run.sh" ]
CMD [ "mongod" ]

```

有了 Dockerfile 文件，还需要一个脚本用于启动的入口。

```

#!/bin/sh
# 容器入口 (PID 1)，以 root 用户身份运行
[ "$1" = "mongod" ] || exec "$@" || exit $?

# 确保数据库属于用户 mongodb
[ "$(stat -c %U /data/db)" = mongodb ] || chown -R mongodb /data/db

# 切换 root 为 mongodb 用户
cmd=exec; for i; do cmd="$cmd '$i'"; done
exec su -s /bin/sh -c "$cmd" mongodb

```

上面的 Dockerfile 文件在启动设置上与官方镜像相似，使用方法基本没有什么变化，因此不再赘述。

14.5 其 他

除了上面介绍的几款数据库，在编程领域还有很多数据库产品，本书不可能全部介绍，实际上 Docker 运行数据库容器都大同小异，只要依赖环境解决了，数据库程序支持在该镜像的基础镜像上运行，那么一个构建一个数据库容器基本不是什么难事。

14.5.1 在容器中使用 SQLite

SQLite 与其他数据库不同的地方是它小到完全不需要镜像，一个文件就是它的全部，使用 Docker 镜像完全是浪费资源。

但是如果在一个容器中使用 SQLite 就不得不提了，因为 SQLite 的特点，不容易设置数据卷（SQLite 保存位置如果受制于应用程序，有时候不能把整个目录挂载上去又不能设置 SQLite 路径），一个比较笨的解决办法就是使用 docker cp 命令，把数据库定时从容器中复制出来。

但是不能实时保存数据库还占用空间，不是一个好办法。另一个办法是使用 Linux 下特有的软链接来实现，首先在容器中创建一个目录，里面存放 SQLite 数据库文件，然后使用软链接把容器内应用程序的数据库文件链接到用户指定目录的数据库文件上，这样容器应用程序使用的实际上是用户指定目录的数据库文件。

如此，用户使用时挂载创建的那个目录即可实现数据持久化。此外，在 PHP 应用中使

用 SQLite 还需要数据库连接的驱动, 不要忘记。

14.5.2 构建自己的 MariaDB 镜像

MariaDB 镜像在使用上与 MySQL 镜像没有什么区别, 因此不单独作为一个大节介绍, 构建 MariaDB 镜像在细节上与 MySQL 有区别所以在本节提及。

这里构建一个基于 Alpine 的 MariaDB 镜像。Dockerfile 文件内容如下:

```
FROM alpine:edge

# 设置数据库默认编码
ENV LC_ALL=en_GB.UTF-8

# 安装 Mariadb 数据库
RUN apk -U upgrade && apk add --no-cache mariadb mariadb-client \
    && mkdir /docker-entrypoint-initdb.d \
    && sed -Ei 's/^(bind-address|log)/#&/ ' /etc/mysql/my.cnf \
    && echo -e 'skip-host-cache\nskip-name-resolve' | awk '{print} $1' \
    && [ "$(mysql --help | grep 'skip-host-cache' | wc -l)" == 0 ] && { c = 1; system("cat") }' /etc/mysql/my.cnf > /tmp/
my.cnf \
    && mv /tmp/my.cnf /etc/mysql/my.cnf
&& rm -rf /tmp/src \
    && rm -rf /var/cache/apk/*

COPY docker-entrypoint.sh /
VOLUME /var/lib/mysql

ENTRYPOINT ["/docker-entrypoint.sh"]
EXPOSE 3306
CMD ["mysql"]
docker-entrypoint.sh 的内容如下 (基于原版修改):
#!/bin/sh

set -e # 错误时终止
set -xo pipefail

DATA_DIR="$(mysql --verbose --help --log-bin-index='mktemp -u' 2>/dev/
null | awk '$1 == "datadir" {print $2; exit}')"
PID_FILE=/run/mysqld/mysqld.pid

# 设置 MYSQL_ROOT_PASSWORD 密码
if [ ! -d "$DATA_DIR/mysql" ]; then
    if [ -z "$MYSQL_ROOT_PASSWORD" -a -z "$MYSQL_ALLOW_EMPTY_PASSWORD" -a -z
"$MYSQL_RANDOM_ROOT_PASSWORD" ]; then
        echo >&2 'error: database is uninitialized and password option is not
specified '
        echo >&2 'You need to specify one of MYSQL_ROOT_PASSWORD, MYSQL_ALLOW_
EMPTY_PASSWORD and MYSQL_RANDOM_ROOT_PASSWORD'
        exit 1
    fi
```

```

# 保证数据库文件夹所有权属于 MySQL
mkdir -p "$DATA_DIR"
chown -R mysql:mysql "$DATA_DIR"

# 初始化数据库
echo 'Initializing database'
mysql_install_db --user=mysql --datadir="$DATA_DIR" --rpm
echo 'Database initialized'

mysqld_safe --pid-file=$PID_FILE --skip-networking --nowatch

mysql_options='--protocol=socket -uroot'

for i in `seq 30 -1 0`; do
    if mysql $mysql_options -e 'SELECT 1' &> /dev/null; then
        break
    fi
    echo 'MySQL init process in progress...'
    sleep 1
done
if [ "$i" = 0 ]; then
    echo >&2 'MySQL init process failed.'
    exit 1
fi

if [ -z "$MYSQL_INITDB_SKIP_TZINFO" ]; then
    apk add --update-cache tzdata
    # sed is for https://bugs.mysql.com/bug.php?id=20545
    mysql_tzinfo_to_sql /usr/share/zoneinfo | sed 's/Local time zone must
be set--see zic manual page/FCTY/' | mysql $mysql_options mysql
fi

if [ ! -z "$MYSQL_RANDOM_ROOT_PASSWORD" ]; then
    MYSQL_ROOT_PASSWORD="$(/dev/urandom tr -dc _A-Z-a-z-0-9 | head -c${1:-10})"
    echo "GENERATED ROOT PASSWORD: $MYSQL_ROOT_PASSWORD"
fi

mysql $mysql_options <<-EOSQL
-- What's done in this file shouldn't be replicated
-- or products like mysql-fabric won't work
SET @@SESSION.SQL_LOG_BIN=0;
DELETE FROM mysql.user ;
CREATE USER 'root'@'%' IDENTIFIED BY '${MYSQL_ROOT_PASSWORD}' ;
GRANT ALL ON *.* TO 'root'@'%' WITH GRANT OPTION ;
DROP DATABASE IF EXISTS test ;
FLUSH PRIVILEGES ;
EOSQL

if [ ! -z "$MYSQL_ROOT_PASSWORD" ]; then
    mysql_options="$mysql_options -p${MYSQL_ROOT_PASSWORD}"
fi

```



```

if [ "$MYSQL_DATABASE" ]; then
    mysql $mysql_options -e "CREATE DATABASE IF NOT EXISTS \"\$MYSQL_
DATABASE\" ;"
    mysql_options="$mysql_options $MYSQL_DATABASE"
fi

if [ "$MYSQL_USER" -a "$MYSQL_PASSWORD" ]; then
    mysql $mysql_options -e "CREATE USER '$MYSQL_USER'@'%' IDENTIFIED BY
'$MYSQL_PASSWORD' ;"

    if [ "$MYSQL_DATABASE" ]; then
        mysql $mysql_options -e "GRANT ALL ON \"\$MYSQL_DATABASE\".* TO '$MYSQL_
USER'@'%' ;"
    fi

    mysql $mysql_options -e 'FLUSH PRIVILEGES ;'
fi

# 执行用户的启动脚本与数据库文件
echo
for f in /docker-entrypoint-initdb.d/*; do
    case "$f" in
        *.sh)     echo "$0: running $f"; . "$f" ;;
        *.sql)    echo "$0: running $f"; mysql $mysql_options < "$f"; echo ;;
        *.sql.gz) echo "$0: running $f"; gunzip -c "$f" | mysql $mysql_options;
echo ;;
        *)       echo "$0: ignoring $f" ;;
    esac
    echo
done

pid="cat $PID_FILE"
if ! kill -s TERM "$pid"; then
    echo >&2 'MySQL init process failed.'
    exit 1
fi

# 确保 MySQL 初始化完全结束
sleep 2

echo
echo 'MySQL init process done. Ready for start up.'
echo
fi

exec mysqld_safe --pid-file=$PID_FILE

```

14.5.3 使用 Docker 部署 Oracle XE 数据库

Oracle XE (Express Edition) 是 Oracle 一个的数据库产品, 本节以该数据库为例介绍 Oracle 数据库产品在 Docker 上的部署。

Oracle XE 的镜像可以在 Docker Hub 中搜索, 这里以 wnameless/oracle-xe-11g 为例 (Ubuntu 16.04 以上):

```
$ docker pull wnameless/oracle-xe-11g
```

或者 (Ubuntu 14.04.4 以下):

```
$ docker pull wnameless/oracle-xe-11g:14.04.4
```

运行容器如下:

```
$ docker run -d -p 2222:22 -p 1521:1521 wnameless/oracle-xe-11g
```

默认参数如下:

```
hostname: localhost
port: 49161
sid: xe
username: system
password: oracle
```

使用 SSH 登录:

```
ssh root@localhost -p 49160
password: admin
```

14.6 本章小结

本章把主流的几个数据库结合实例全部介绍了一遍, 还重新构建了这几款数据库镜像, 更细致地了解了数据库镜像的构成。

本章只是对数据库容器的使用作了基础的实战演示, 实际上使用 Docker 配合其他组件可以轻松搭建高可用的数据库集群以及主从复制等任务, 具体的实战例子在后面会深入讲解。

第 15 章 编程语言

流行的编程语言有很多，本章选取了较有代表性的几种编程语言为例，介绍如何使用与构建这些基础镜像。

当学到这里，基本上每位读者都可以熟练地使用 Docker 的基本指令，即使面对一些稍微复杂或者配置烦琐的镜像，相信大家凭着传统 Linux 操作经验也能很快定位问题并解决，但秉着“少踩坑”的念头，本章选取了几个经典的应用作为实战例子。

为了更好地学习本章内容，读者不妨建立一个自己的 Dockerfile 仓库托管在 Github 上，以便更好地学习本章内容。

通过本章的学习，读者将掌握对编程语言的镜像构建，可以更好地使用 Docker 来开发。本章的知识点有：

- 剖析编程语言的官方镜像。
- 学习构建基于 Alpine 和 Debian/Ubuntu 不同版本的基础镜像。
- 区分 Dev 与 Production 两种镜像构建模式。
- 语言框架应用实战。

15.1 C/C++ 语言

C/C++语言是当前最流行也最高效的开发语言之一，有大量的软件依靠 C/C++运行环境。

15.1.1 官方镜像 library/gcc

因为 Docker 从 Linux 发展而来，首先想到的自然就是 GCC 了，官方镜像地址为 https://hub.docker.com/_/gcc/，作为一个开源的语言编译器，GCC（GNU Compiler Collection，GNU 编译器套件）是由 GNU 开发的编程语言编译器，它是以 GPL 许可证所发行的自由软件，也是 GNU 计划的关键部分。

使用官方镜像可以使用 `docker pull gcc` 命令拉取最新版本的 GCC，目前最新版本的是 6.2.0，如图 15.1 所示。

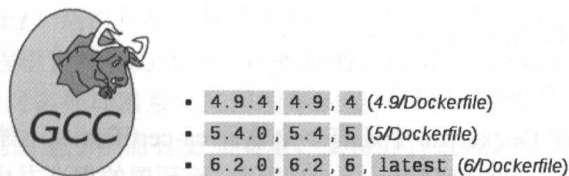


图 15.1 GCC 版本与 GCC 标志

GCC 的镜像可以做的事情太多了，这里以一个 Hello World 举例，首先新建一个文件夹，名称为 `myapp`，然后在 `myapp` 里面新建一个 `main.c` 文件，内容为输出一行文字：

```
#include<stdio.h>
int main() {
    printf("Hello World\n");
    return 0;
}
```

最后在 `myapp` 文件夹的同级目录下新建一份 `Dockerfile`。

```
FROM gcc:4.9
COPY . /myapp
WORKDIR /myapp
RUN gcc -o /myapp main.c
CMD ["/myapp"]
```

然后只需要构建执行，就可以看到输出一行 Hello World 了。

```
$ docker build -t my-gcc-app .
$ docker run -it --rm --name my-running-app my-gcc-app
```

除了把源代码放到容器里面编译的方法，还可以挂载数据卷编译本地项目，这样既保证了项目的保密性，也保证了编译结果的安全性。

依旧以上面的 `myapp` 为例，这一次不复制到容器内：

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4.9 gcc -o myapp myapp.c
```

这样就可以编译一个本地项目了。

面对简单的项目，可以通过上面的方法来构建，但是很多时候一个项目往往有不少依赖情况，这个时候使用 Docker 镜像时，可以像平时在 Linux 宿主机命令行下操作一样，使用 `make` 来构建，所以在 Docker 中也一样。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4.9 make
```

使用这句命令的前提是项目文件夹内有 `Makefile` 文件。

15.1.2 基于 Alpine 构建 C/C++ 镜像

在 15.1.1 节的例子中，当编译一个简单的文件时却要拉取一个几百 MB 的镜像，有种杀鸡用牛刀的感觉，所以自然会想到精简 GCC 镜像。而说到精简镜像，首先会想到我们在第 11 章中讲过的操作系统 Alpine。

所以本节内容将使用 Alpine 作为基础镜像构建一个 GCC 镜像，在第 11 章中介绍过的 Alpine 目前的 latest 版本是 3.4，这里也是用该版本构建。

一个最简单的例子就是安装 `libstdc++`，以下是 Alpine 的移植库：

```
FROM alpine

RUN apk update && apk upgrade \
    && apk add ca-certificates libstdc++ \
    && rm -rf /var/cache/apk/*
```

除了 `libstdc++`，在 `Dockerfile` 中还有一个软件 `ca-certificates`，对于各个 Linux 发行版来说证书相关的密码和认证的机制由 `openssl` 提供，而预置的根证书由 `ca-certificates` 提供，`ca-certificates` 包由 `debian` 维护，主页在 <http://packages.debian.org/sid/ca-certificates>，所以这

是我们加入 `ca-certificates` 的原因。然后使用 `docker build` 构建如下：

```
$ docker build -t yourname/gcc .
Sending build context to Docker daemon 5.632 kB
Step 1 : FROM alpine
--> ee4603260daa
Step 2 : RUN apk update && apk upgrade && apk add ca-certificates libstdc++
&& rm -rf /var/cache/apk/*
--> Running in dfdb2a8cd2c6
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/community/x86_64/APKINDEX.tar.gz
v3.4.4-14-g3b19e7e [http://dl-cdn.alpinelinux.org/alpine/v3.4/main]
v3.4.4-12-gebd7753 [http://dl-cdn.alpinelinux.org/alpine/v3.4/community]
OK: 5973 distinct packages available
(1/2) Upgrading libcrypto1.0 (1.0.2i-r0 -> 1.0.2j-r0)
(2/2) Upgrading libssl1.0 (1.0.2i-r0 -> 1.0.2j-r0)
Executing busybox-1.24.2-r11.trigger
OK: 5 MiB in 11 packages
(1/3) Installing ca-certificates (20160104-r4)
(2/3) Installing libgcc (5.3.0-r0)
(3/3) Installing libstdc++ (5.3.0-r0)
Executing busybox-1.24.2-r11.trigger
Executing ca-certificates-20160104-r4.trigger
OK: 7 MiB in 14 packages
--> 80b87a3d2267
Removing intermediate container dfdb2a8cd2c6
Successfully built 80b87a3d2267
```

从 `docker images` 中可以看到镜像的大小不到 10 MB。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
zuolan/gcc	latest	80b87a3d2267	3 minutes ago	9.767 MB

这个镜像可以运行大部分一般的 C/C++ 程序，但是这个镜像没有包含更多的编译构建工具，如果要执行编译动作，可以在 `Dockerfile` 中加入相关软件包。

例如，下面是一份带 C++ 开发环境的 Alpine 镜像。

```
FROM alpine

RUN apk update && apk upgrade \
    && apk add ca-certificates build-base boost \
    && rm -rf /var/cache/apk/*
```

在 Alpine 中，所有基本的构建工具都被包含在一个叫做 `build-base` 的软件包里面，使用它可以执行一些常见的基本构建动作。Boost 是一个扩展的 C++ 函数库。

本节的代码可以在 <https://github.com/izuolan/dockerfiles/tree/master/base/gcc> 找到。

15.2 Golang 语言

Go 语言 (Golang) 是由谷歌主导开发的编程语言，语法清晰、设计精良、性能出色，成为当下服务端编程语言的有力竞争者，本书的主角 Docker 就是使用 Go 语言编写的。

Go 语言专门针对多处理器系统应用程序的编程进行了优化，使用 Go 编译的程序可以媲美 C 或 C++ 代码的速度，而且更加安全、支持并行进程。

15.2.1 官方镜像 library/golang

Golang 的镜像一样可以从官网找到，地址为 https://hub.docker.com/_/golang/。

使用 `docker pull golang` 可以拉取最新版的 Golang 镜像。如图 15.2 所示为目前 Golang 镜像的版本详情，可以注意到 Go 语言还有一个 `windowsservercore` 的版本，这是专门为 Windows Server 2016 准备的镜像。

```

• 1.6.3, 1.6 (1.6/Dockerfile)
• 1.6.3-onbuild, 1.6-onbuild (1.6/onbuild/Dockerfile)
• 1.6.3-wheezy, 1.6-wheezy (1.6/wheezy/Dockerfile)
• 1.6.3-alpine, 1.6-alpine (1.6/alpine/Dockerfile)
• 1.6.3-windowsservercore, 1.6-windowsservercore
  (1.6/windows/windowsservercore/Dockerfile)
• 1.7.1, 1.7, 1, latest (1.7/Dockerfile)
• 1.7.1-onbuild, 1.7-onbuild, 1-onbuild, onbuild (1.7/onbuild/Dockerfile)
• 1.7.1-wheezy, 1.7-wheezy, 1-wheezy, wheezy (1.7/wheezy/Dockerfile)
• 1.7.1-alpine, 1.7-alpine, 1-alpine, alpine (1.7/alpine/Dockerfile)
• 1.7.1-windowsservercore, 1.7-windowsservercore, 1-windowsservercore,
  windowsservercore (1.7/windows/windowsservercore/Dockerfile)

```

图 15.2 Go 语言在 Docker Hub 的版本情况

从官网的 Dockerfile 来看，Go 语言的安装非常简单，只需要解压设置环境变量即可，以 `golang 1.6` 为例，镜像基于 `buildpack-deps:jessie-scm`，这是一个安装了很多基础工具的镜像，主要安装了 `curl`、`wget`、`bzip`、`git`、`mercurial`、`openssh-client`、`subversion`、`procps` 等工具。

```
FROM buildpack-deps:jessie-scm
```

接下来安装 C/C++ 编译环境，因为 Go 语言开发中还有一些场景需要依赖 C/C++ 开发环境。

```

# gcc for cgo
RUN apt-get update && apt-get install -y --no-install-recommends \
    g++ \
    gcc \
    libc6-dev \
    make \
    pkg-config \
    && rm -rf /var/lib/apt/lists/*

```

设置 Go 语言版本、下载链接等信息，方便修改、升级、维护。

```

ENV GOLANG_VERSION 1.6.3
ENV GOLANG_DOWNLOAD_URL https://golang.org/dl/go$GOLANG_VERSION.linux-
amd64.tar.gz
ENV GOLANG_DOWNLOAD_SHA256 cdde5e08530c0579255d6153b08fdb3b8e47caabbe717
bc7bcd7561275a87aeb

```

解压安装清理。

```

RUN curl -fsSL "$GOLANG_DOWNLOAD_URL" -o golang.tar.gz \
    && echo "$GOLANG_DOWNLOAD_SHA256 golang.tar.gz" | sha256sum -c - \
    && tar -C /usr/local -xzf golang.tar.gz \
    && rm golang.tar.gz

```

设置 Go 语言变量。

```
ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH
RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH"
WORKDIR $GOPATH
COPY go-wrapper /usr/local/bin/
```

上面是对基于 Debian 构建的 Go 语言镜像的 Dockerfile 解析,除了这个 tag,官方还提供了其他不少版本,很多时候可以根据标签判断这个镜像的作用,比如 Go 镜像中的 golang:onbuild 标签,是专门为构建环境而创建的镜像,与 golang:<version>不同的地方在于, onbuild 包含了更完善的构建环境,当然也意味着体积更大。

如果要使用 Docker 镜像开发一个 Go 应用,推荐使用 golang:1.6-onbuild 标签,在第一部分讲解 Dockerfile 的章节有提到一个 ONBUILD 的指令, golang:onbuild 的 Dockerfile 就用到了该指令,一共三句,意味着在使用 FROM golang:1.6-onbuild 的方式构建新应用时,会执行 ONBUILD 的三句命令,这时用户应该注意把自己的项目源代码放置在 Dockerfile 同级目录下,而 go-wrapper 是官方的一个 shell 脚本,作用类似 go get -d -v 和 go install -v。

```
FROM golang:1.6

RUN mkdir -p /go/src/app
WORKDIR /go/src/app

# this will ideally be built by the ONBUILD below ;)
CMD ["go-wrapper", "run"]

ONBUILD COPY . /go/src/app
ONBUILD RUN go-wrapper download
ONBUILD RUN go-wrapper install
```

使用如下构建命令构建应用。

```
$ docker build -t my-golang-app .
$ docker run -it --rm --name my-running-app my-golang-app
```

很多时候,会出于代码保密性的要求等,不适合把源代码放进一个容器运行传输(目前官方没有给出容器加密的方案,但是从最近启用 <https://store.docker.com/> 域名来看, Docker 正在尝试镜像收费、加密等新功能),这时候需要在本地编译一个 Go 项目。

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.6 go build -v
```

-w 指定工作目录, -v 指定数据卷位置,如果包含 Makefile,可以使用 make 命令:

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.6 bash -c make
```

除了上面的情况,还有一种 Go 特有的特性就是跨平台编译,这一点在 Docker 上同样可以做到。比如在 Linux 系统下编译 Windows 的软件,可以使用:

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp -e GOOS=windows -e GOARCH=386 golang:1.6 go build -v
```

甚至可以同时构建多个平台的应用:

```
$ docker run --rm -it -v "$PWD":/usr/src/myapp -w /usr/src/myapp golang:1.6 bash
$ for GOOS in darwin linux; do
>   for GOARCH in 386 amd64; do
>     go build -v -o myapp-$GOOS-$GOARCH
>   done
> done
```


主要的环境变量就两个，即 GOOS 与 GOARCH，分别设置系统与架构。

15.2.2 Beego 框架

Beego 是 Go 语言下一个非常著名的 Web 框架，国人开发，文档完善，深受国内开发者们欢迎，所以我们不妨使用 Docker 构建一个 Beego 的镜像作为实战例子。

如图 15.3 所示为两者的 Logo，Beego 官网为 <http://beego.me/>。

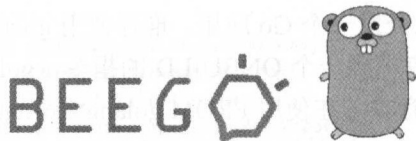


图 15.3 Beego 与 Golang

常规下，安装 Beego 只需要两句命令就可以安装一个 Beego 开发环境了。

```
$ go get github.com/astaxie/beego
$ go get github.com/beego/bee
```

所以同样的，在 Docker 中也可以如此轻松地构建 Beego 镜像。

```
FROM golang:1.6-onbuild
```

```
RUN go get -u -v github.com/astaxie/beego && go get -u -v github.com/beego/bee
RUN cd $GOPATH/src && \
bee new hello
WORKDIR $GOPATH/src/hello
CMD ["bee", "run", "hello"]
```

构建之后，运行时可以在 <http://localhost:8080/> 看到一个 Hello World 的页面，说明 Beego 已经成功运行。

15.2.3 自助 Git 服务——Gogs

Gogs 是一款极易搭建的自助 Git 服务，界面如图 15.4 所示。Gogs 同样是国人开发的、超人气的明星应用，Gogs 在 Docker Hub 的地址是 <https://hub.docker.com/r/gogs/gogs/>，由官方维护，与 Github 仓库同步更新，用户可以直接拉取该镜像运行。

Gogs 支持 Sqlite，对于个人用户，Sqlite 足以应付，所以硬件资源有限的情况下不妨选择 Sqlite。

Gogs 默认端口是 3000，于是打算把 git.example.com 解析到 3000 端口。

这里需要 Nginx 做反代理，所以新建一个 nginx 文件夹，然后新建一份 gogs.conf，内容如下：

```
server
{
    listen 80;
    server_name git.example.com; # 这里填自定义域名
    location / {
        proxy_redirect off;
        proxy_set_header Host $host;
```

```
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_pass http://172.17.0.3:3000; # 这里填容器地址，如果不知道的话填写
公网 IP 也可以
}
```

然后启动两个容器：

```
$ docker run -it \
  --name=gogs \
  -p [3022]:22 -p [3000]:3000 \
  -v [/path/data]:/data gogs/gogs
$ docker run -it \
  --name=nginx \
  -p 80:80 \
  -v [~/nginx/]:/etc/nginx/conf.d/ nginx:alpine
```



图 15.4 Gogs 界面

如读者所见“[]”里面的内容自己决定，可以自由修改。默认容器叫做 gogs，数据保存在 [/path/data]，端口是 3000 和 3022。

- -p 3022:22 会将 3022 暴露给外网，用于 Git 的 SSH 协议，不用 SSH 可以去掉。
- -v [/path/data]:/data 将映射本地的 [/path/data] 目录作为 volume 给容器使用，根据自己创建的目录修改此项。

然后检查一下容器有没有正常在运行：

```
$ docker ps
```

访问网站进行初次的配置，配置中需要注意的是以下两项不用修改。

- Repository Root Path（仓库根目录）；
- Run User（运行用户）。

其他可根据用户的情况修改，Nginx 已经帮我们把容器内的 3000 端口在服务器上代理给 80 端口了。

如果需要启动一个带有 SSL 证书的 Git 仓库，可以使用 Caddy Server 做反代，新建一个文件夹 Caddy，然后在里面新建一个 Caddyfile，内容如下：

```
git.example.com {
    proxy / 123.456.789.0:3000 { # 改成你 IP:port
        proxy_header Host {host}
        proxy_header X-Real-IP {remote}
        proxy_header X-Forwarded-Proto {scheme}
    }
    log /var/log/caddy.log
    gzip
    tls i@example.com # 你的邮箱
}
```

然后启动 Caddy 容器：

```
$ docker run -it \
    --name=caddy \
    -p 80:80 -p 443:443 \
    -v [~/Caddy/Caddyfile]:/etc/Caddyfile abiosoft/caddy
```

这样启动之后的 Caddy 容器会使用 Letsencrypt 的证书为 Gogs 服务提供 HTTPS，关于 Caddy 的内容在前面服务器章节有介绍。更详细的 Gogs 部署教程可以阅读 <https://zuolan.me/p/docker-gogs.html>。

Gogs 的 Docker 镜像还有一个树莓派版本 <https://hub.docker.com/r/gogs/gogs-rpi/>。

15.2.4 基于 Alpine 构建 Golang 镜像

如同上面 C/C++ 的镜像一样，很多时候 Golang 的镜像都显得非常臃肿，而官方也给出了 Alpine 构建的镜像，地址为 <https://github.com/docker-library/golang/blob/master/1.6/alpine/Dockerfile>。

不过在 Alpine 活跃的社区中，有用户对 Golang 做了打包，所以可以直接像安装普通软件一样安装 Golang 环境。

```
FROM alpine:edge

RUN echo '@edge http://nl.alpinelinux.org/alpine/edge/main' >> /etc/apk/
repositories
RUN echo '@community http://nl.alpinelinux.org/alpine/edge/community' >>
/etc/apk/repositories

RUN apk update && apk upgrade
RUN apk add ca-certificates curl make git bzr mercurial go@community
alpine-sdk

RUN rm -rf /var/cache/apk/*

RUN mkdir /go
ENV GOPATH /go
```

使用 docker build 命令构建即可，alpine-sdk 软件包含了许多基础软件，因此可以胜任绝大部分的 Go 开发。本节的 Dockerfile 可以在 <https://github.com/izuolan/dockerfiles/tree/master/base/go> 找到源代码。

15.3 Java 语言

Java 语言是一种计算机编程语言，拥有跨平台、面向对象、泛型编程的特性，广泛应用于企业级 Web 应用开发和移动应用开发。Java 作为一门盘踞在诸多编程语言排行榜首位的编程语言，必然有着它的魅力。

15.3.1 官方镜像 library/openjdk

因为 Oracle JDK 在法律上比较敏感的原因，Docker Hub 上已经停止对 Oracle JDK（https://hub.docker.com/_/java/）的维护，转而维护社区出身的 openjdk（https://hub.docker.com/_/openjdk/），在本书中会对两种 JDK 都进行讲解，如图 15.5 所示为 OpenJDK 在 Docker Hub 的版本。

- 6b38-jdk, 6b38, 6-jdk, 6 (6-jdk/Dockerfile)
- 6b38-jre, 6-jre (6-jre/Dockerfile)
- 7u111-jdk, 7u111, 7-jdk, 7 (7-jdk/Dockerfile)
- 7u91-jdk-alpine, 7u91-alpine, 7-jdk-alpine, 7-alpine (7-jdk/alpine/Dockerfile)
- 7u111-jre, 7-jre (7-jre/Dockerfile)
- 7u91-jre-alpine, 7-jre-alpine (7-jre/alpine/Dockerfile)
- 8u102-jdk, 8u102, 8-jdk, 8, jdk, latest (8-jdk/Dockerfile)
- 8u92-jdk-alpine, 8u92-alpine, 8-jdk-alpine, 8-alpine, jdk-alpine, alpine (8-jdk/alpine/Dockerfile)
- 8u102-jre, 8-jre, jre (8-jre/Dockerfile)
- 8u92-jre-alpine, 8-jre-alpine, jre-alpine (8-jre/alpine/Dockerfile)
- 9-b139-jdk, 9-b139, 9-jdk, 9 (9-jdk/Dockerfile)
- 9-b139-jre, 9-jre (9-jre/Dockerfile)

图 15.5 OpenJDK 在 Docker Hub 的版本

从 Github 仓库的 Dockerfile 来看，openjdk 镜像都是通过软件仓库直接安装的，所以就不在这里分析 Dockerfile 的指令了。而使用 openjdk 镜像也非常简单，例如要运行一个 Java 应用，可以直接新建一个 Dockerfile，内容如下：

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

然后构建镜像并运行：

```
$ docker build -t my-java-app .
$ docker run -it --rm --name my-running-app my-java-app
```

也可以像前面的例子一样构建本地项目：

```
$ docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp openjdk:7 javac Main.java
```

15.3.2 基于 Alpine 构建 Java 镜像

Oracle JDK 和 Open JDK 的镜像在使用上并无差别,在 15.3.1 节中 library/openjdk 为例,这里以 Oracle JDK 为例,构建一个定制的基于 Alpine 的 Java 镜像。

Java 和前面遇到的语言环境不同,稍微复杂一些,在构建之前先确定 JDK 的依赖,使用 Java 肯定需要 JRE,而 JRE 需要 glibc 的支持,所以只需要确保依赖满足,JRE 程序就可以正常运行了。

新建 Dockerfile 如下:

```
FROM alpine:edge
# 设置环境变量
ENV JAVA_VERSION_MAJOR=8 \
    JAVA_VERSION_MINOR=73 \
    JAVA_VERSION_BUILD=02 \
    JAVA_PACKAGE=server-jre \
    GLIBC_PKG_VERSION=2.23-r1 \
    LANG=en_US.UTF8

# 设置工作目录,建议为/tmp,方便清理工作
WORKDIR /tmp

# 这里用到 https://github.com/andyshinn/alpine-pkg-glibc/ 的 glibc 库
# 通过 curl 设置 Cookie 来获取 Oracle JDK 的压缩包,建议阅读官网的协议书
RUN apk add --no-cache --update-cache curl ca-certificates bash && \
    curl -Lo /etc/apk/keys/andyshinn.rsa.pub "https://github.com/andyshinn/alpine-pkg-glibc/releases/download/${GLIBC_PKG_VERSION}/andyshinn.rsa.pub" && \
    curl -Lo glibc-${GLIBC_PKG_VERSION}.apk "https://github.com/andyshinn/alpine-pkg-glibc/releases/download/${GLIBC_PKG_VERSION}/glibc-${GLIBC_PKG_VERSION}.apk" && \
    curl -Lo glibc-bin-${GLIBC_PKG_VERSION}.apk "https://github.com/andyshinn/alpine-pkg-glibc/releases/download/${GLIBC_PKG_VERSION}/glibc-bin-${GLIBC_PKG_VERSION}.apk" && \
    curl -Lo glibc-il8n-${GLIBC_PKG_VERSION}.apk "https://github.com/andyshinn/alpine-pkg-glibc/releases/download/${GLIBC_PKG_VERSION}/glibc-il8n-${GLIBC_PKG_VERSION}.apk" && \
    apk add glibc-${GLIBC_PKG_VERSION}.apk glibc-bin-${GLIBC_PKG_VERSION}.apk glibc-il8n-${GLIBC_PKG_VERSION}.apk && \
    curl -jksSLH "Cookie: oraclelicense=accept-securebackup-cookie" \
    "http://download.oracle.com/otn-pub/java/jdk/${JAVA_VERSION_MAJOR}u${JAVA_VERSION_MINOR}-b${JAVA_VERSION_BUILD}/${JAVA_PACKAGE}-${JAVA_VERSION_MAJOR}u${JAVA_VERSION_MINOR}-linux-x64.tar.gz" | gunzip -c - | tar -xf - && \
    apk del curl ca-certificates && \
    mv jdk1.${JAVA_VERSION_MAJOR}.0_${JAVA_VERSION_MINOR}/jre /jre && \

# 清理,减少镜像体积
rm /jre/bin/jjs && \
rm /jre/bin/keytool && \
rm /jre/bin/orbd && \
rm /jre/bin/pack200 && \
rm /jre/bin/policytool && \
rm /jre/bin/rmid && \
rm /jre/bin/rmiregistry && \
rm /jre/bin/servertool && \
```

```

rm /jre/bin/tnameserv && \
rm /jre/bin/unpack200 && \
rm /jre/lib/ext/nashorn.jar && \
rm /jre/lib/jfr.jar && \
rm -rf /jre/lib/jfr && \
rm -rf /jre/lib/oblique-fonts && \
rm -rf /tmp/* /var/cache/apk/* && \
echo 'hosts: files mdns4_minimal [NOTFOUND=return] dns mdns4' >>
/etc/nsswitch.conf

# 设置全局变量
ENV JAVA_HOME=/jre
ENV PATH=${PATH}:${JAVA_HOME}/bin

```

执行构建，镜像大小约 130.4 MB。如图 15.6 所示为 OpenJDK 与 Oracle JDK 的 Logo。



图 15.6 OpenJDK 与 Oracle JDK

此外，如果 OpenJDK 能够满足需求的话，也可以使用下面的 Dockerfile 来构建，构建之后的大小为 102.6 MB。

```

FROM alpine

RUN echo '@edge http://nl.alpinelinux.org/alpine/edge/main' >> /etc/apk/
repositories && \
    echo '@community http://nl.alpinelinux.org/alpine/edge/community' >>
/etc/apk/repositories && \
    apk update && \
    apk upgrade && \
    apk add ca-certificates openjdk8-jre-base@community && \
    apk del ca-certificates && \
rm -rf /tmp/* /var/cache/apk/*

```

15.3.3 Tomcat 服务器

谈到 Java 的 Web 服务器，Tomcat 必定是绕不开的著名开源产品。Tomcat 是 Apache 软件基金会（Apache Software Foundation）Jakarta 项目中的一个核心项目。Tomcat 服务器是一个免费开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试 JSP 程序的首选。

在 Docker Hub 上也有官方镜像在维护地址为 https://hub.docker.com/_/tomcat/。

运行一个 Tomcat 服务器非常轻松，只需要执行如下命令：

```
$ docker run -it --rm tomcat:8.0
```

一个 Tomcat 容器就启动了，通过 <http://container-ip:8080> 可以访问初始页面。Tomcat 的 Dockerfile 最后一句 CMD ["catalina.sh", "run"] 说明了用户在使用过程中如果覆盖启动指令时，需要带上脚本名称，例如：


```
$ docker run -it --rm tomcat:8.0 sh -c 'catalina.sh stop'
```

Tomcat 的镜像提供了一些环境变量可以在启动时指定（Tomcat 版本 7 和 8）：

```
CATALINA_BASE:    /usr/local/tomcat
CATALINA_HOME:    /usr/local/tomcat
CATALINA_TMPDIR:  /usr/local/tomcat/temp
JRE_HOME:         /usr
CLASSPATH:
/usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar
```

下面是 Tomcat 6 的环境变量

```
CATALINA_BASE:    /usr/local/tomcat
CATALINA_HOME:    /usr/local/tomcat
CATALINA_TMPDIR:  /usr/local/tomcat/temp
JRE_HOME:         /usr
CLASSPATH:        /usr/local/tomcat/bin/bootstrap.jar
```

通过 -e 参数改变镜像以适应用户的应用。配置文件在 /usr/local/tomcat/conf/ 目录下。

15.3.4 下一代集成开发环境——Eclipse Che

Eclipse Che 是在 Codenvy 公司的开源 IDE 基础上开发的新一代 Eclipse，运行在浏览器里，使用 Docker 作为工作引擎，目前正处于频繁的开发阶段，界面如图 15.7 所示，本节以在 Eclipse Che 里开发 Android 应用作为例子，展示 Docker 与 Java 的强强结合。

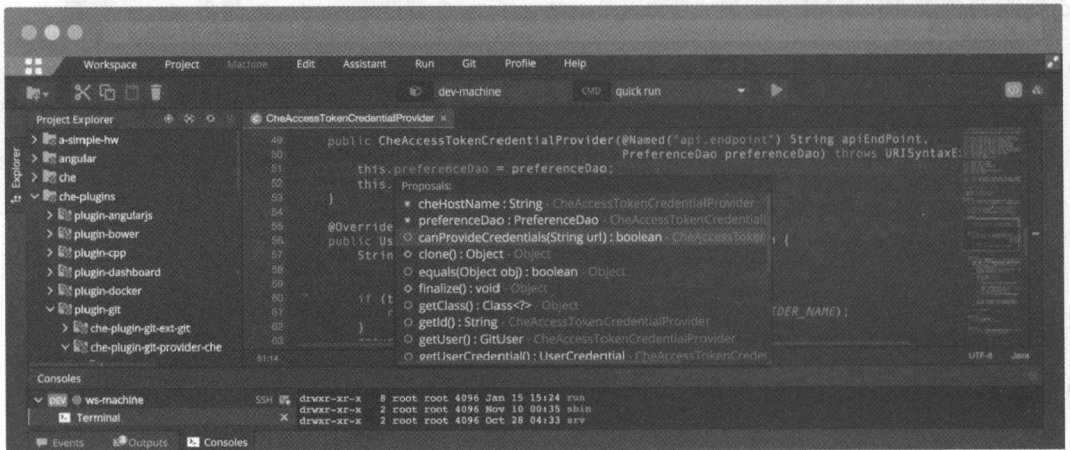


图 15.7 Eclipse Che 界面

启动 Eclipse Che 只需一句命令：

```
$ docker run --rm -t -v /var/run/docker.sock:/var/run/docker.sock
eclipse/che start
# YOUR OUTPUT
INFO: -----
INFO: ECLIPSE CHE: CONTAINER STARTING
INFO: ECLIPSE CHE: SERVER BOOTING
INFO: ECLIPSE CHE: See logs at "docker logs -f che"
INFO: ECLIPSE CHE: BOOTED AND REACHABLE
INFO: ECLIPSE CHE: http://<your-che-host>:8080
INFO: -----
2016-10-15 20:46:24,204[main] [INFO ] [o.a.catalina.startup.Catalina 642]
- Server startup in 9052 ms
```


出现上面最后一句时表示容器启动成功，此时打开浏览器，输入 localhost:8080 可以看到后台面板。

更多命令如下：

```
$ docker run --rm -t \
-v /var/run/docker.sock:/var/run/docker.sock eclipse/che [COMMAND]
# 可选命令：
start      # 启动 Che 服务
stop       # 停止 Che 服务
restart    # 重启 Che 服务
update     # 拉取最新版本的 Che 镜像
info       # 打印调试信息
```

启动之后，新建一个工作区（Workspace），然后选择一个示例并新建项目，这里以 Android 为例，选择新建一个 Android 项目，如图 15.8 所示。

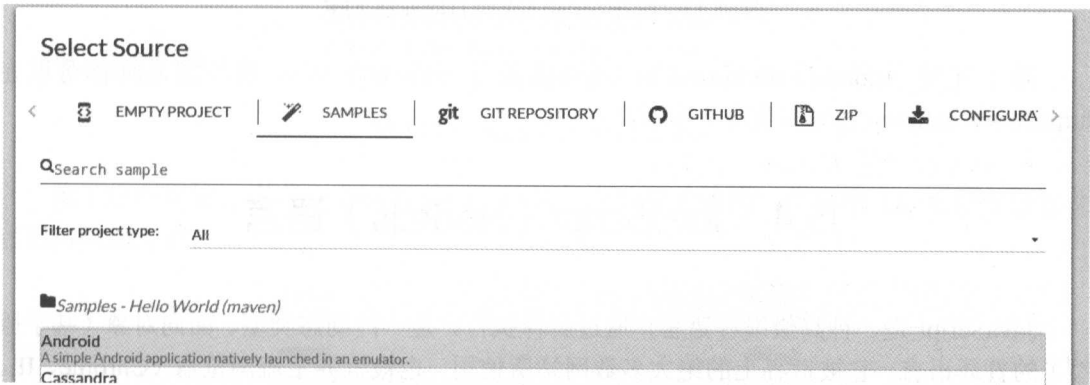


图 15.8 选择示例并新建 Android 项目

选择一个镜像启动，默认为 codenvy/ubuntu_android，镜像体积因为包含了 Android 开发的环境，体积很大，如图 15.9 所示为目前可用的 Android 镜像版本，用户可以自行构建，在图 15.10 中可以看到镜像的 Dockerfile 与其他信息。

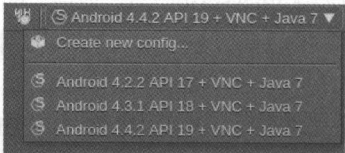


图 15.9 选择一个镜像启动

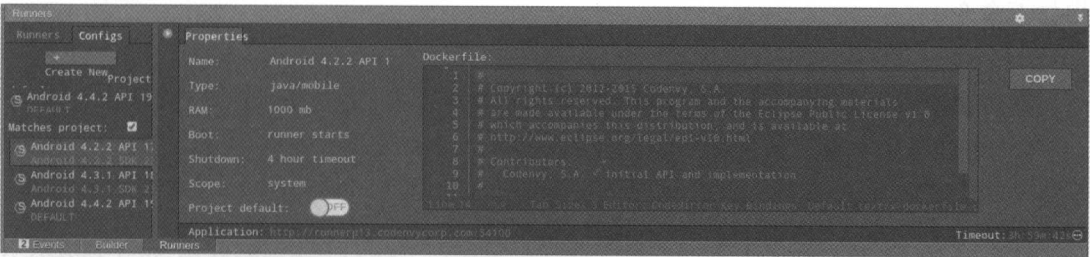


图 15.10 configs 命令可以查看 Dockerfile 文件信息

在浏览器开发 Android，APP 界面是通过 VNC 远程显示的，如图 15.11 所示，需要比较先进的浏览器 Chrome 或者 Firefox 等。

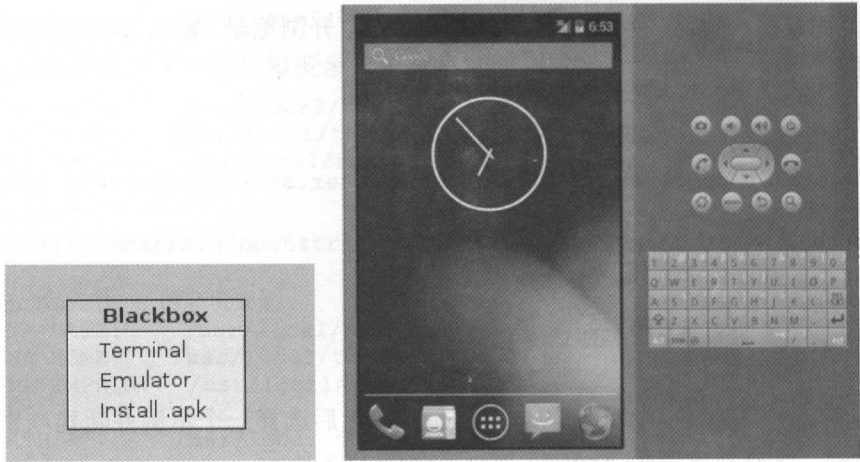


图 15.11 选择 Install.apk 会自动安装 APK

除了开发 Android，Eclipse Che 还可以进行一系列的 Web 端开发，具体可以在 <http://www.eclipse.org/che/> 获得更多信息。

15.4 JavaScript (Node.js) 语言

JavaScript 是一种高级编程语言，通过解释执行，是一门动态类型、面向对象（基于原型）的直译语言。它被世界上的绝大多数网站所使用，也被世界主流浏览器（Chrome、IE、FireFox、Safari、Opera）支持。

Node.js 是一个开放源代码、跨平台的、可用于服务器端和网络应用的运行环境。Node.js 应用 JavaScript 语言写成。Node.js 提供事件驱动和非阻塞 I/O API，可优化应用程序的吞吐量和规模。这些技术通常被用于实时应用程序。Node.js 采用 Google 的 V8 引擎来执行代码。

15.4.1 官方镜像 library/node

Docker Hub 中的 Library 仓库有 Node.js 的镜像（https://hub.docker.com/r/_/node/），使用如下命令：

```
$ docker pull node
```

即可拉取，目前 Docker Hub 上的 Node.js 页面共有图 15.12 所示的这些标签。

图 15.12 中的标签大致分类可以分为两大类，一类是生产环境的标签（不带 `onbuild` 或者带 `slim` 标签），一类是开发环境的标签（带有 `onbuild`），使用 `onbuild` 标签的镜像表示其内部安装了一系列构建开发工具，或者在生产环境的镜像之上添加了 `ONBUILD` 指令，而生产环境中的镜像一般情况下只需要 Node.js 环境即可运行，不需要使用 `onbuild` 标签的镜像。

至于为什么有 0.1 版本开头的标签，相信了解 Node.js 历史的人都知道两者的区别，因此在使用上可根据需要选择适合的标签。



- 7.0.0, 7.0, 7, latest (7.0/Dockerfile)
- 7.0.0-onbuild, 7.0-onbuild, 7-onbuild, onbuild (7.0/onbuild/Dockerfile)
- 7.0.0-slim, 7.0-slim, 7-slim, slim (7.0/slim/Dockerfile)
- 7.0.0-wheezy, 7.0-wheezy, 7-wheezy, wheezy (7.0/wheezy/Dockerfile)
- 6.9.1, 6.9, 6, boron (6.9/Dockerfile)
- 6.9.1-onbuild, 6.9-onbuild, 6-onbuild, boron-onbuild (6.9/onbuild/Dockerfile)
- 6.9.1-slim, 6.9-slim, 6-slim, boron-slim (6.9/slim/Dockerfile)
- 6.9.1-wheezy, 6.9-wheezy, 6-wheezy, boron-wheezy (6.9/wheezy/Dockerfile)
- 4.6.1, 4.6, 4, argon (4.6/Dockerfile)
- 4.6.1-onbuild, 4.6-onbuild, 4-onbuild, argon-onbuild (4.6/onbuild/Dockerfile)
- 4.6.1-slim, 4.6-slim, 4-slim, argon-slim (4.6/slim/Dockerfile)
- 4.6.1-wheezy, 4.6-wheezy, 4-wheezy, argon-wheezy (4.6/wheezy/Dockerfile)
- 0.12.17, 0.12, 0 (0.12/Dockerfile)
- 0.12.17-onbuild, 0.12-onbuild, 0-onbuild (0.12/onbuild/Dockerfile)
- 0.12.17-slim, 0.12-slim, 0-slim (0.12/slim/Dockerfile)
- 0.12.17-wheezy, 0.12-wheezy, 0-wheezy (0.12/wheezy/Dockerfile)
- 0.10.48, 0.10 (0.10/Dockerfile)
- 0.10.48-onbuild, 0.10-onbuild (0.10/onbuild/Dockerfile)
- 0.10.48-slim, 0.10-slim (0.10/slim/Dockerfile)
- 0.10.48-wheezy, 0.10-wheezy (0.10/wheezy/Dockerfile)

图 15.12 Node.js 在 Docker Hub

除了这些标签，还有一个标签 slim，这是一个精简的镜像，从如图 15.13 中可以看出它们体积的区别。



图 15.13 不同标签的体积区别

图 15.13 中，3 个镜像的 Dockerfile 中最大的不同在于 FROM 的内容，例如 7.0.0 标签：
FROM buildpack-deps:jessie

而 7.0.0-slim 标签中的 FROM 内容如下：

FROM buildpack-deps:jessie-curl

再看 7.0.0-onbuild 的 Dockerfile 内容如下：

FROM node:7.0.0

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

ONBUILD ARG NODE_ENV

ONBUILD ENV NODE_ENV \$NODE_ENV

ONBUILD COPY package.json /usr/src/app/

```
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app

CMD [ "npm", "start" ]
```

可以看到 `onbuild` 标签是基于 7.0.0 的标签构建的，通常情况下构建一个 Node.js 镜像可以使用以下方式构建。

```
FROM node:7.0.0-onbuild
```

在 <https://github.com/nodejs/docker-node/> 中的 Dockerfile 全部是使用 `curl` 从官网拉取安装包安装的，本节以另一种方式介绍构建 Node.js 镜像，也就是从 `apt` 安装 Node.js：

```
RUN curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash - \
    && apt-get install -y nodejs
```

上面的链接中修改 6.x 为相应的版本号，例如 4.x 就可以安装 Node.js 4.0。从 5.0 版本开始，Node.js 就已经默认安装 `npm`，如果在使用过程中发现没有安装 `npm`，可以使用下面的方式快速给镜像安装 `npm` 环境。

```
RUN curl http://npmjs.org/install.sh | sh
# 设置 npm 仓库
# RUN npm config set registry https://registry.npm.taobao.org \
    && npm config get registry
```

如果上面的 `onbuild` 镜像不符合应用要求，还可以构建一个自己的 Node.js 生产环境镜像。

```
FROM node:latest

LABEL "version"="7.0.0"
ENV NODE_PATH=/usr/local/lib/node_modules/:/usr/local/lib NODE_ENV=production
RUN npm install -g node-gyp \
    && mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY . /usr/src/app
ONBUILD RUN npm install

EXPOSE 8080
CMD [ "npm", "start" ]
```

然后使用 `docker build` 命令来构建一个自己的生产环境镜像。

```
$ docker build -t user/node-pro .
```

最后，可以在此基础上构建应用了，只需要在 Node.js 项目目录下新建 Dockerfile，并且使用 `user/node-pro` 作为基础镜像。

```
FROM user/node-pro
```

如此即可构建该应用。

15.4.2 vue.js 框架

近年来，前端领域各种框架层出不穷，这些前端工具在配置使用上经常需要用到 Node.js 环境，本节以 Vue 为例，介绍前端框架、工具在 Docker 中的应用。

Vue.js 是一个前端框架，Vue.js 通过简洁的 API 提供高效的数据绑定和灵活的组件系统，使其在前端纷繁复杂的生态中受到一定程度的关注。

```
FROM node:slim
RUN npm install -g vue-cli@2.4.0
ENTRYPOINT ["vue"]
CMD ["-h"]
```

使用 `build` 命令构建:

```
$ docker build -t vue-cli .
```

构建完成后可以使用下面的方式使用 `vue` 了。

```
$ docker run --rm -ti \
  -v $PWD:/srv \
  -w /srv \
  rflavien/vue-cli init <template-name> <project-name>
```

为了更方便操作, 我们还可以使用 `alias` 来设置别名:

```
alias vue='docker run --rm -ti -v $PWD:/srv -w /srv rflavien/vue-cli'
```

这样就可以通过 `vue init <template-name> <project-name>` 来创建项目了, 输入 `vue` 就相当于原生体验 `vue` 命令一样。

通过这个例子, 还可以举一反三把这个思路应用到其他 Node.js 工具中去。

15.4.3 Express 框架

Express 是一种保持最低程度规模的灵活 Node.js Web 应用程序框架, 为 Web 和移动应用程序提供一组强大的功能。

本节使用 Express 作为例子介绍 Node.js 框架在 Docker 上的应用。与之前一样, 构建使用 Docker 镜像前, 需要一份 Dockerfile, 本例子中 Express 的 Dockerfile 不算复杂。

```
# 基于 node:slim 镜像
FROM node:slim
# 从当前目录复制文件到 /www 目录
COPY ./ /www
# 设定工作目录为 /www
WORKDIR /www
# 安装应用依赖
RUN npm set progress=false && \
  npm install --global --progress=false
# 暴露端口
EXPOSE 3000
# 设置启动命令
CMD ["node", "app.js"]
```

在指定项目目录中, 保存文件为 `Dockerfile`, 然后使用 `docker build` 构建可以获得该 Express 项目的应用镜像。

在这个例子中可以看到, 其实 `Dockerfile` 中没有任何与 Express 有关的指令, 却可以构建一个 Express 应用, 这是因为项目的配置是保存在项目中, 只要有项目配置文件, `npm install` 自然会解决所有依赖。

换句话说, 不只是 Express 项目, 其他 Node.js 应用, 同样可以使用这份 `Dockerfile` 去构建, 这与 Node.js 的特性是分不开的。

15.4.4 浏览器里的 IDE——Cloud9-IDE

前面介绍了构建 Node.js 工具和框架后，这里再介绍一个使用 Docker 部署 Node.js 应用的例子。

Cloud9 IDE 是一个开源的在线集成开发环境，与前面介绍的 Eclipse Che 类似，Cloud9 IDE 使用 Node.js 开发，编辑器支持语法提示、多人协作等特点。简单来说就是一个在浏览器中的 IDE，界面如图 15.14 所示。

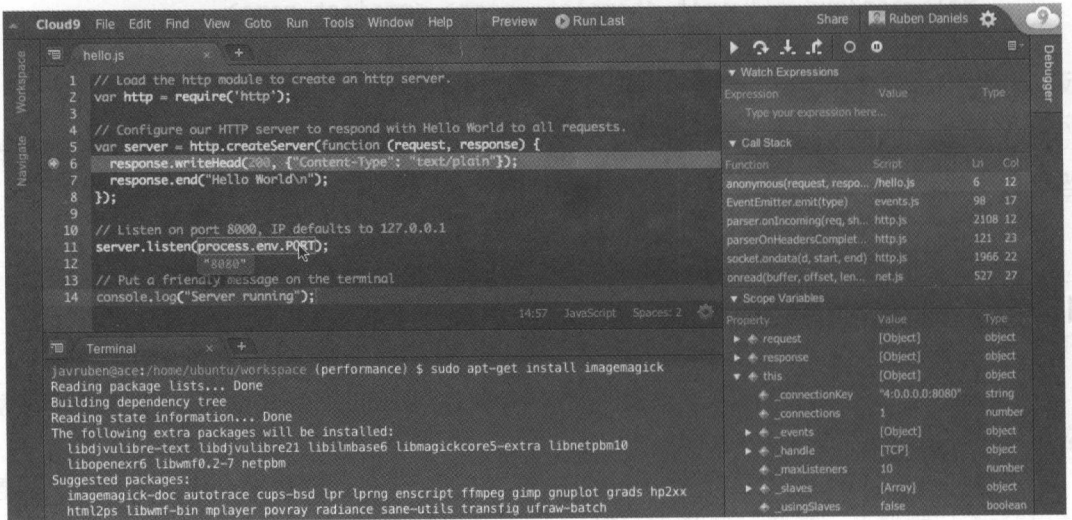


图 15.14 Cloud9 IDE 界面

本节以部署 Cloud9 IDE 为例子，首先在 Cloud9 IDE 的 Github 页面获取安装信息。Github 地址为 <https://github.com/c9/core>。

在 README 中提示有安装脚本，因此使用它来安装 Cloud9 IDE 即可。新建一个文件夹名为 cloud9-ide，然后在文件夹里面新建文件 Dockerfile，然后选定一个基础镜像，本例使用 ubuntu:trusty 作为基础镜像。

```
FROM ubuntu:trusty
```

不使用 node 的镜像是为了减少镜像体积。

然后尝试确定 Cloud9 IDE 安装过程的软件依赖，有时候这些不是一次就能确定的，需要通过多次构建，逐步定位软件的依赖，一般来说需要 make、build-essential 这两个软件包，为了方便操作，可以定义一个构建依赖变量 buildDeps，以及一个软件依赖变量 softDeps。

```
ARG buildDeps='make build-essential g++ gcc python curl ca-certificates' \
softDeps='nodejs git'
```

之所以要分开，是因为构建依赖变量中的软件在完成 Cloud9 IDE 安装之后是要卸载的，而软件依赖是 Cloud9 IDE 运行时必须要使用的软件，因此这样分开十分容易管理，在以后的 Dockerfile 中，读者也可以尝试使用这样的方式管理软件依赖。

确定软件依赖之后就是安装了，使用 --no-install-recommends 参数可以减少安装不必要的软件包。首先安装构建依赖。


```
RUN apt-get update && apt-get upgrade -y \
  && apt-get install -y $buildDeps --no-install-recommends
然后是软件依赖:
RUN curl -sL https://deb.nodesource.com/setup | sudo bash - \
  && apt-get install -y $softDeps
```

完成依赖安装之后, 还需要一个 **forever** 工具来管理 Node.js 进程, 这个工具可以使用 **npm** 安装。

```
RUN npm install -g forever && npm cache clean
```

完成这些之后就可以安装 Cloud9 IDE 了, 从仓库克隆代码然后使用脚本自动安装, 最后不要忘记卸载构建依赖以及清理缓存。

```
RUN git clone https://github.com/c9/core.git /cloud9 && cd /cloud9 \
  && scripts/install-sdk.sh \
  && apt-get purge -y --auto-remove $buildDeps \
  && apt-get autoremove -y && apt-get autoclean -y && apt-get clean -y \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
  && npm cache clean
```

Cloud9 IDE 需要使用一个目录作为它的工作目录, 本例中使用 **/workspace** 作为工作目录, 目录的指定实际上是通过 **ENTRYPOINT** 指令的 **forever** 来指定的。

```
VOLUME /workspace
ENV $workspace /workspace
EXPOSE 8181
ENTRYPOINT ["forever", "/cloud9/server.js", "-w", "/workspace", "-l",
"0.0.0.0"]
```

```
# CMD["--auth", "username:password"]
```

根据 Cloud9 IDE 的 API, 可以在 **ENTRYPOINT** 后面接上更多参数, 例如 **CMD** 的注释一样为 Cloud9 IDE 开启访问验证。

为了精简镜像体积与结构, 完整的 **Dockerfile** 如下:

```
FROM ubuntu:trusty
```

```
RUN buildDeps='make build-essential g++ gcc python curl ca-certificates'
&& softDeps="nodejs git" \
  && apt-get update && apt-get upgrade -y \
  && apt-get install -y $buildDeps --no-install-recommends \
  && curl -sL https://deb.nodesource.com/setup | sudo bash - \
  && apt-get install -y $softDeps \
  && npm install -g forever && npm cache clean \
  && git clone https://github.com/c9/core.git /cloud9 && cd /cloud9 \
  && scripts/install-sdk.sh \
  && apt-get purge -y --auto-remove $buildDeps \
  && apt-get autoremove -y && apt-get autoclean -y && apt-get clean -y \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
  && npm cache clean
```

```
VOLUME /workspace
ENV $workspace /workspace
EXPOSE 8181
ENTRYPOINT ["forever", "/cloud9/server.js", "-w", "/workspace", "-l",
"0.0.0.0"]
```

```
#CMD["--auth", "username:password"]
```

使用构建命令构建:


```
$ docker build -t cloud9-ide .
```

然后使用 `docker run` 命令运行，此时可以看到 Cloud9 IDE 的界面如图 15.15 所示。

```
$ docker run -d -v $(pwd):/workspace -p 8181:8181 zuolan/cloud9-ide
```

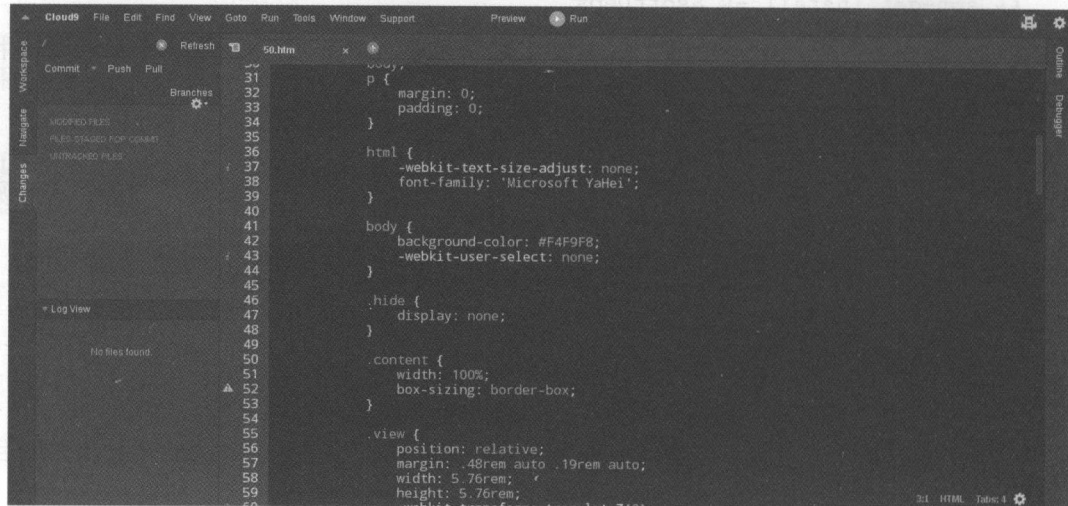


图 15.15 Cloud9 IDE 运行在 Docker 上

启用访问验证启动 `cloud9-ide`。

```
$ docker run -d \
  -v $(pwd):/workspace \
  -p 8181:8181 \
  zuolan/cloud9-ide --auth username:password
```

本节源码在 Github 上可以找到，地址为 <https://github.com/izuolan/dockerfiles/blob/master/cloud9-ide/>。

15.4.5 基于 Alpine 构建 Node.js 镜像

按照本书的惯例，自然是还要使用 Alpine 来构建一个 Node.js 镜像。在 Alpine 仓库中已经有 Node.js 的软件包，因此构建起来非常方便：

非开发模式的 Node.js 镜像 Dockerfile，包含 Node.js 运行环境与依赖。

```
FROM alpine:edge
RUN echo '@edge http://nl.alpinelinux.org/alpine/edge/main' >> /etc/apk/
repositories \
  && apk update && apk upgrade \
  && apk add ca-certificates nodejs@edge \
  && npm uninstall -g npm \
  && rm -rf /var/cache/apk/*
```

开发模式的 Node.js 镜像 Dockerfile，除了 Node.js 运行环境，还包含开发工具。

```
FROM alpine:edge
RUN echo '@edge http://nl.alpinelinux.org/alpine/edge/main' >> /etc/apk/
repositories \
  && apk update && apk upgrade \
  && apk add ca-certificates git nodejs-dev@edge nodejs@edge \
  && npm install -g npm \
  && rm -rf /var/cache/apk/*
```

源代码在 Github 上可以找到, 地址为 <https://github.com/izuolan/dockerfiles/blob/master/base/node>。

15.5 PHP 语言

PHP 是一种开源的通用计算机脚本语言, 尤其适用于网络开发并可嵌入 HTML 中使用。PHP 的语法借鉴吸收 C 语言、Java 和 Perl 等流行计算机语言的特点, 易于一般程序员学习。

PHP 的主要目标是允许网络开发人员快速编写动态页面, 但 PHP 也被应用于其他很多领域中。PHP 的应用范围相当广泛, 尤其是在网页程序的开发上。一般来说 PHP 大多运行在网页服务器上, 通过运行 PHP 代码来产生用户浏览的网页。

15.5.1 官方镜像 library/php

PHP 在 Docker Hub 上的标签非常多, 如图 15.16 所示, 主要分为基于 Debian 和 Alpine 两大类, 每个版本都基于两种镜像构建两个版本, 每个版本又根据 fpm、apache、zts 分为几个标签。它们虽然标签不同, 但是用法基本一致。关于 fpm 与 zts 的区别不在本书讨论范围, 用户可根据自己需要选择。

- 7.1.0RC4-cli, 7.1-rc-cli, rc-cli, 7.1.0RC4, 7.1-rc, rc (7.1-rc/Dockerfile)
- 7.1.0RC4-alpine, 7.1-rc-alpine, rc-alpine (7.1-rc/alpine/Dockerfile)
- 7.1.0RC4-apache, 7.1-rc-apache, rc-apache (7.1-rc/apache/Dockerfile)
- 7.1.0RC4-fpm, 7.1-rc-fpm, rc-fpm (7.1-rc/fpm/Dockerfile)
- 7.1.0RC4-fpm-alpine, 7.1-rc-fpm-alpine, rc-fpm-alpine (7.1-rc/fpm/alpine/Dockerfile)
- 7.1.0RC4-zts, 7.1-rc-zts, rc-zts (7.1-rc/zts/Dockerfile)
- 7.1.0RC4-zts-alpine, 7.1-rc-zts-alpine, rc-zts-alpine (7.1-rc/zts/alpine/Dockerfile)
- 7.0.12-cli, 7.0-cli, cli, 7.0.12, 7.0, 7, latest (7.0/Dockerfile)
- 7.0.12-alpine, 7.0-alpine, alpine (7.0/alpine/Dockerfile)
- 7.0.12-apache, 7.0-apache, apache (7.0/apache/Dockerfile)
- 7.0.12-fpm, 7.0-fpm, fpm (7.0/fpm/Dockerfile)
- 7.0.12-fpm-alpine, 7.0-fpm-alpine, fpm-alpine (7.0/fpm/alpine/Dockerfile)
- 7.0.12-zts, 7.0-zts, zts (7.0/zts/Dockerfile)
- 7.0.12-zts-alpine, 7.0-zts-alpine, zts-alpine (7.0/zts/alpine/Dockerfile)
- 5.6.27-cli, 5.6-cli, 5-cli, 5.6.27, 5.6, 5 (5.6/Dockerfile)
- 5.6.27-alpine, 5.6-alpine, 5-alpine (5.6/alpine/Dockerfile)
- 5.6.27-apache, 5.6-apache, 5-apache (5.6/apache/Dockerfile)
- 5.6.27-fpm, 5.6-fpm, 5-fpm (5.6/fpm/Dockerfile)
- 5.6.27-fpm-alpine, 5.6-fpm-alpine, 5-fpm-alpine (5.6/fpm/alpine/Dockerfile)
- 5.6.27-zts, 5.6-zts, 5-zts (5.6/zts/Dockerfile)
- 5.6.27-zts-alpine, 5.6-zts-alpine, 5-zts-alpine (5.6/zts/alpine/Dockerfile)



图 15.16 PHP 的版本

一般而言, 编程语言的镜像都是带有基础镜像色彩的, 它们实际上是为应用提供一个运行环境, 因此一般使用这些镜像来构建相应的应用镜像。使用 PHP 镜像构建应用非常简单, 例如 Dockerfile 如下, 即可构建一个 PHP 应用。

```
FROM php:7.0-cli
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
CMD [ "php", "./your-script.php" ]
```

Dockerfile 把当前项目目录的文件复制到容器的目录中，然后使用 PHP 执行，这是一个非常基础的应用。实际情况中很少会遇到这么简单的 PHP 应用，如果要运行单个 PHP 文件，也不需要重新构建镜像，使用数据卷挂载即可。

```
$ docker run -it --rm --name my-running-script \
-v "$PWD":/usr/src/myapp \
-w /usr/src/myapp php:7.0-cli \
php your-script.php
```

PHP 应用中大部分是网络应用，需要用浏览器来访问操作。因此通常需要配合 Web 服务器，与 PHP 搭配多年的好搭档 Apache 自然是不错的选择，使用 php:7.0-apache 标签即可，该标签的镜像已经包含 PHP 和 Apache 两个环境。例如，使用 php:7.0-apache 标签来构建一个网络应用。

```
FROM php:7.0-apache
COPY src/ /var/www/html/
```

根据 COPY 的特点，源代码应当放在 src/ 目录中，COPY 会把源代码复制到 /var/www/html/ 之中，然后 Apache + PHP 会解析启动并显示它们。

```
$ docker build -t my-php-app .
$ docker run -d --name my-running-app my-php-app
```

使用过 PHP 的读者大都认识 php.ini 文件，在 Docker 中也可以使用自定义的配置，在 Docker 镜像中，配置文件的位置在 /usr/local/etc/php 中，例如，下面复制本地配置文件到镜像中构建。

```
FROM php:7.0-apache
COPY config/php.ini /usr/local/etc/php/
COPY src/ /var/www/html/
```

从该例中我们知道，php:7.0-apache 镜像中两个关键的位置其实可以作为数据卷来挂载，可以不需要构建镜像，但需要注意的是 /usr/local/etc/php 这个目录非空，因此使用数据卷时应当仅挂载 php.ini 一个文件。

```
$ docker run -d -p 80:80 \
--name my-apache-php-app \
-v php.ini /usr/local/etc/php/php.ini:ro \
-v "$PWD":/var/www/html php:7.0-apache
```

与其他 Library 仓库的编程语言镜像一样，PHP 镜像也是从源代码开始构建的，这样的好处自然不必多言，但是往往很耗时间。如果需要定制构建自己的 PHP 环境，可以选择从软件仓库直接安装相应版本的 PHP，例如使用 apt-get 安装 PHP。

```
FROM ubuntu:14.04
```

```
# 安装 PHP 5 环境
RUN apt update && apt install \
    imagemagick \
    php5-cli \
    php5-fpm \
    php5-json \
    php5-intl \
    php5-curl \
    php5-mysqldb \
    php5-xdebug \
    php5-memcached \
    php5-mcrypt \
```

```

php5-gd \
php5-sqlite \
php5-xmlrpc \
php5-xsl \
php5-geoip \
php5-ldap \
php5-memcache \
php5-memcached \
php5-imagick \
php-pear \
&& pear channel-update pear.php.net \
&& pear upgrade-all \
&& pear config-set auto_discover 1 \
&& ln -sf /etc/php5/mods-available/mcrypt.in /etc/php5/cli/conf.d/
20-mcrypt.ini \
&& ln -sf /etc/php5/mods-available/mcrypt.in /etc/php5/fpm/conf.d/
20-mcrypt.ini \
&& curl -sS https://getcomposer.org/installer | php -- --install-dir=
/usr/local/bin/ --filename=composer

```

CMD [“php” , “-a”]

15.5.2 快速安装扩展

PHP 的一个让人着迷的特点就是扩展丰富易用，在 Docker 中的 PHP 镜像依旧把这一优良传统保留了下来，Docker 镜像提供了 3 个工具用于安装配置扩展，分别是 `docker-php-ext-configure`、`docker-php-ext-install`、`docker-php-ext-enable`。

为了保持镜像体积小巧，PHP 的源码保存在一个压缩的 tar 文件中。为了方便 PHP 扩展与 PHP 源代码链接，`docker-php-source` 可以轻松从压缩包中提取或者删除源代码（如果不需要那么直接删除即可）。例如：

```

FROM php:7.0-apache
RUN docker-php-source extract \
    # 这里执行你的代码 \
    && docker-php-source delete

```

上面的方法不失为一种办法，但实际上我们安装的扩展大部分都已经打包，因此使用 `docker-php-ext-install` 工具安装会更加容易，例如，要在 PHP 镜像中安装 `iconv`、`mcrypt` 和 `gd` 这 3 个扩展，可以使用 `docker-php-ext-install` 工具。

```

FROM php:7.0-fpm
RUN apt-get update && apt-get install -y \
    libfreetype6-dev \
    libjpeg62-turbo-dev \
    libmcrypt-dev \
    libpng12-dev \
    && docker-php-ext-install -j$(nproc) iconv mcrypt \
    && docker-php-ext-configure gd --with-freetype-dir=/usr/include/
--with-jpeg-dir=/usr/include/ \
    && docker-php-ext-install -j$(nproc) gd

```

`docker-php-ext-configure` 用于配置插件。

除了这些开源扩展，有些 PHP 扩展没有提供源代码，但是提供了 PECL，这时安装一个 PECL 扩展可以使用 `pecl install` 命令，然后使用 `docker-php-ext-enable` 来启用它们：

```
FROM php:7.0-fpm
RUN apt-get update && apt-get install -y libmemcached-dev \
    && pecl install memcached \
    && docker-php-ext-enable memcached
```

有些 PHP 扩展甚至也没有 PECL，这时候安装它们就需要使用 RUN 命令来安装，然后使用 docker-php-ext-enable 来启用。

```
FROM php:7.0-apache
RUN curl -fsSL 'https://xcache.lighttpd.net/pub/Releases/3.2.0/xcache-3.2.0.tar.gz' -o xcache.tar.gz \
    && mkdir -p xcache \
    && tar -xf xcache.tar.gz -C xcache --strip-components=1 \
    && rm xcache.tar.gz \
    && ( \
        cd xcache \
        && phpize \
        && ./configure --enable-xcache \
        && make -j$(nproc) \
        && make install \
    ) \
    && rm -r xcache \
    && docker-php-ext-enable xcache
```

15.5.3 LNMP 环境组合

我们常说的 LNMP 或者 LAMP 都是 PHP 开发常用的组合，在传统的开发环境中，要手动部署一个 LNMP 环境不仅操作烦琐，还容易出错，不同的系统之间迁移起来十分困难，但是有了 Docker，一切变得简单明了，再也不需要动手安装配置 PHP 环境、Nginx 服务器、MySQL 数据库等，一句话就可以部署整个 LNMP 服务。

在学习 Docker 的早期，很多人都习惯把 Docker 当做一个类似虚拟机一样的工具，在一个容器里面安装 PHP、Nginx、MySQL 等全部工具，不仅把镜像体积堆积得特别大，不利于管理，还有一个更严重的问题就是根本不符合 Docker 一个容器一个进程的理念，换句话说，这样把全部工具丢到一个镜像里面去构建的做法，是完全体现不出容器技术的优点的。这样做不仅增加了部署难度，还增加了备份难度。

那么回到正题，还记得前面详细讲过的 Docker Compose 吧？我们可以构建几个镜像，然后使用 Docker Compose 把它们连接起来，本节会用到这个工具。

首先新建一个文件夹(本例名为 lnmp)，在文件夹里新建一个文件 docker-compose.yml，内容如下：

```
# Web 服务器
nginx:
  image: nginx:alpine
  ports:
    - "80:80"
  volumes:
    # 应用位置
    - /path/your/app:/usr/share/nginx/html/
    # nginx 配置
    - /path/your/conf.d:/etc/nginx/conf.d:ro
  links:
    - fpm: __DOCKER_PHP_FPM__
```

```
# php-fpm
fpm:
  image: php:7.0-fpm
  ports:
    - "9000:9000"
  volumes:
    - /path/your/app:/usr/share/nginx/html
  links:
    - mysql:mysql
```

```
# 数据库
mysql:
  image: mysql:5.6
  ports:
    # 映射端口 3306 到本地 3306 端口
    - "3306:3306"
  volumes:
    - /path/your/mysql:/var/lib/mysql
  environment:
    - MYSQL_ROOT_PASSWORD=YOUR_PASSWORD
```

上面就是一份最简单的 LNMP 的 Compose 配置文件，使用下面命令即可启动：

```
$ docker-compose up -d
```

现在拆分来看看这个配置文件有些什么内容，首先是十分清晰的 3 个服务，各司其职。

Web 服务器上，我们使用的是 nginx:alpine 镜像，服务内容如下：

```
nginx:
  image: nginx:alpine
  ports:
    - "80:80"
    - "443:443"
  volumes:
    # 应用目录
    - /home/zuolan/docker/app:/usr/share/nginx/html/
    # nginx 配置目录
    - /home/zuolan/docker/conf.d:/etc/nginx/conf.d:ro
    # 证书数据卷
    - ./server.crt:/etc/nginx/server.crt:ro
    - ./server.key:/etc/nginx/server.key:ro
  links:
    # 设置容器连接
    - fpm: __DOCKER_PHP_FPM__
```

上面对原来的 Compose 配置文件做了些修改，加入了 SSL 的配置。conf.d 里面的文件内容如下：

```
server {
    listen 80;
    listen 443 ssl;
    ssl_certificate      /etc/nginx/server.crt;
    ssl_certificate_key  /etc/nginx/server.key;
    ssl_protocols        TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers           HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers on;

    server_name localhost;

    #charset koi8-r;
    #access_log /var/log/nginx/log/host.access.log main;
```

```

location / {
    root    /usr/share/nginx/html;
    index  index.html index.htm index.php;
}
.....
location ~ \.php$ {
    root            html;
    fastcgi_pass    __DOCKER_PHP_FPM__:9000;
    fastcgi_index   index.php;
    fastcgi_param   SCRIPT_FILENAME /usr/share/nginx/html$fastcgi_script_
name;
    include         fastcgi_params;
}
.....
}

```

Docker 环境并不会影响 Nginx 的配置，因此这些配置信息相信大家都十分熟悉（完整模板在本节后面有链接）。

然后再来看 PHP 服务，该服务选择的是 `php:7.0-fpm` 镜像。

```

fpm:
  image: php:7.0-fpm
  ports:
    - "9000:9000"
  volumes:
    - /docker/app:/usr/share/nginx/html
    # 你的 php.ini 文件
    - /docker/php/php.ini-production:/usr/local/etc/php/php.ini:ro
  links:
    - mysql:mysql

```

`php.ini` 的内容根据自己的需求修改即可。使用 `--link` 之后，数据库的变量会进入 PHP 服务容器。

最后是数据库服务，这部分没有什么可以扩展的，相关变量在数据库章节已经全部介绍过了。

使用 `Compose` 命令启动：

```
$ docker-compose up -d
```

这样就已经部署了 LNMP 一整套服务，相当方便。

本节 Github 源码地址为 <https://github.com/izuolan/dockerfiles/tree/master/lnmp>

15.5.4 基于 Alpine 构建 PHP 镜像

根据本书的惯例，自然是不会忘记基于 Alpine 构建镜像的实战，本节将基于 Alpine 构建 PHP 镜像做讲解。本节选择的是 PHP 7.0 版本作为例子，PHP 7.0 在 Alpine 仓库目前还处于测试阶段，如果需要部署一个稳定版本的 PHP，可以参考 <https://github.com/izuolan/dockerfiles/tree/master/base/php/php5> 链接中的 Dockerfile 构建。

Alpine 安装软件与其他发行版一样，直接指定软件名即可，如果是测试源，要加

@testing 符号。

```
FROM alpine:edge

RUN echo '@testing http://nl.alpinelinux.org/alpine/edge/testing' >> /etc/
apk/repositories \
  && apk update && apk upgrade \
  && apk add ca-certificates \
    php7 \
    php7-openssl \
    php7-phar \
    php7-json \
    php7-curl \
  && rm -rf /var/cache/apk/* \
  && ln -s /usr/bin/php7 /usr/bin/php
```

然后使用 **docker build** 来构建：

```
$ docker build -t user/php:alpine .
```

与 Node 一样，为构建的镜像添加更丰富的开发环境，例如安装 **php7-dev** 而不是 **php7**：

```
FROM alpine:edge

RUN apk update && apk upgrade \
  && apk add ca-certificates curl php7-dev@testing \
  && curl -sS https://getcomposer.org/installer | php \
  && mv composer.phar /usr/local/bin/composer \
  && apk del curl && rm -rf /var/cache/apk/*
```

15.5.5 自建私有云存储——ownCloud

介绍了如何在 Docker 下快速部署 LNMP 之后，现在再来看如何部署一个 PHP 应用，相信很多人都需要一个不限速、质量好、有保障、传输加密的网盘。

既然这样不如自己部署一个吧，ownCloud 就是一个 PHP 网盘应用。为了更便捷地启用 HTTPS，本节依旧选择 **caddy** 作为 Web 服务器。

首先使用命令新建文件夹 **caddy** 与文件 **Caddyfile**。

```
$ cd && mkdir caddy && cd caddy && vim Caddyfile
```

然后复制粘贴下面内容到 **Caddyfile** 中：

```
example.com {
  proxy / 233.233.233.233:2333 {
    proxy_header Host {host}
    proxy_header X-Real-IP {remote}
    proxy_header X-Forwarded-Proto {scheme}
  }
  log /var/log/caddy.log
  gzip
}
```

其中 **example.com** 替换为用户的域名，**233.233.233.233** 替换为用户的服务器 IP，**2333** 替换为想要设置的端口（默认为 2333）。

然后再新建一个文件 `docker-compose.yml`:

```
$ cd && mkdir owncloud && cd owncloud && vim docker-compose.yml
```

然后复制粘贴下面内容到 `docker-compose.yml` 文件中(上面 2333 如果改成其他端口了, 下面的 2333 也要相应改成那个端口; ownCloud 支持 Sqlite 和 MySQL, 所以这里有两份配置文件, 按需取用)。

Sqlite 版本如下:

```
owncloud:
  image: owncloud
  volumes:
    - ~/cloud/config:/opt/owncloud/config
    - ~/cloud/data:/opt/owncloud/data
  ports:
    - 2333:80

caddy:
  image: abiosoft/caddy
  volumes:
    - ~/caddy/Caddyfile:/etc/Caddyfile
    - ~/.caddy:/root/.caddy
  ports:
    - 80:80
    - 443:443
```

MySQL 版本如下:

```
version: '2'
services:
  db:
    container_name: db
    image: mysql:5.7
    volumes:
      - "~/mysql:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 这里填密码
      MYSQL_DATABASE: owncloud
      MYSQL_USER: 这里填数据库用户名
      MYSQL_PASSWORD: 这里还是填密码
  owncloud:
    container_name: owncloud
    depends_on:
      - db
    image: owncloud:latest
    volumes:
      - ~/cloud/config:/var/www/html/config
      - ~/cloud/data:/var/www/html/data
    links:
      - db
    ports:
      - "2333:80"
    restart: always
```

如果使用 MySQL 安装，数据库地址是 db（不是 localhost），数据库名称是 ownCloud。
最后，使用 `docker-compose up -d` 启动，没什么问题的话，稍等片刻即可完成安装并启动，打开浏览器输入地址可以看到如图 15.17 所示的界面，设置管理员账号信息之后就可以使用了。

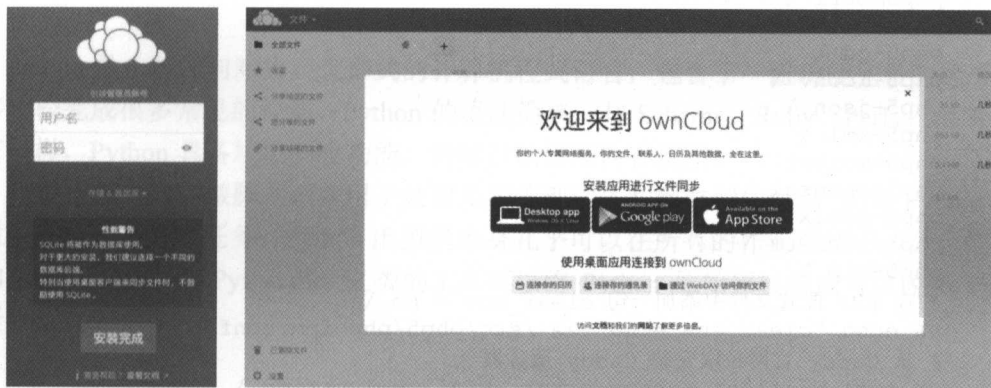


图 15.17 ownCloud 部署成功

注意，在 Centos 6 下使用 Docker 的官方脚本安装 Docker 会默认使用软件源里的 Docker，该软件源默认 Docker 是 1.9 版本，因此无法使用 Docker Compose 启动，老版本用户需要手动启动容器：

```
# 运行 ownCloud，端口 2333 可以修改为用户喜欢的，注意和第一步的 Caddyfile 一致
$ docker run -v ~/cloud/config:/opt/owncloud/config \
  -v ~/cloud/data:/opt/owncloud/data \
  -p 2333:80 \
  -d --name=owncloud owncloud

# 运行 Caddy。
$ docker run -v ~/caddy/Caddyfile:/etc/Caddyfile \
  -v ~/.caddy:/root/.caddy \
  -p 80:80 -p 443:443 \
  --name=caddy -d abiosoft/caddy
```

15.5.6 Typecho 博客系统

有了前面基于 Alpine 构建 PHP 的经验，不妨用 Dockerfile 来构建一个 PHP 应用，本节选择一个在国内比较有名的开源博客程序——Typecho。

在本例的镜像中，不使用 Nginx 作为 Web 服务器，而是使用 Caddy 作为 Web 服务器，构建一个 Typecho 镜像。

Dockerfile 内容如下：

```
FROM alpine:3.4
MAINTAINER Zuo Lan <i@zuolan.me>

# 在 Alpine 中安装 PHP 环境
RUN apk add --update --no-cache \
  ca-certificates \
  openssl \
  php5-fpm \
  sqlite \
```

```

curl \
git \
php5-pdo \
php5-pdo_sqlite \
php5-sqlite3 \
php5-ctype \
php5-curl \
php5-dom \
php5-gd \
php5-iconv \
php5-json \
php5-xml \
php5-mcrypt \
php5-openssl \
php5-posix \
php5-sockets \
php5-zip \
# 在 PHP 配置文件中添加一句 clear_env = no \
&& echo "clear_env = no" >> /etc/php5/php-fpm.conf \
# 从 Caddy 官网拉取安装 Caddy 服务器 \
&& curl --silent --show-error --fail --location \
--header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \
"https://caddyserver.com/download/build?os=linux&arch=amd64" \
| tar --no-same-owner -C /usr/bin/ -xz caddy \
&& chmod 0755 /usr/bin/caddy \
&& /usr/bin/caddy -version

WORKDIR /srv
EXPOSE 80 443 2015
ADD Caddyfile /etc/caddy/Caddyfile
VOLUME ["/srv/", "/etc/caddy/"]

ENTRYPOINT ["/usr/bin/caddy"]
CMD ["--conf", "/etc/caddy/Caddyfile"]

```

其中 Caddyfile 的内容如下:

```

0.0.0.0
browse
fastcgi / 127.0.0.1:9000 php
startup php-fpm
log stdout
errors stdout

```

Caddyfile 的语法不在本书讨论范围,有兴趣的读者可以在 Caddy 文档中阅读更多信息。
完成上述操作,使用:

```
$ docker build -t typecho
```

即可构建 Typecho 镜像。

运行时先把 Typecho 代码从 Github 上拉取到本地:

```
$ git clone https://github.com/typecho/typecho.git ~/app/
$ chmod 777 ~/app/usr && chmod 777 ~/app
```

然后使用 Docker 运行:

```
$ docker run -p 80:80 \
-p 443:443 \
-v ~/app/:/srv/ \
-v ~/caddy/:/etc/caddy/ \
-d typecho
```

~/app/表示本地 Typecho 源代码的目录。

15.6 Python 语言

Python 是一种面向对象、直译式的计算机程式语言，包含了一组功能完备的标准库，能够轻松完成很多常见的任务。Python 的语法简单，与 Scheme、Ruby、Perl、Tcl 等动态语言一样，Python 具备垃圾回收功能，能够自动管理内存使用。

Python 经常被当做脚本语言用于处理系统管理任务和网路程式编写，但是 Python 也非常适合完成各种高阶任务。Python 虚拟机本身几乎可以在所有的作业系统中运行，使用一些如 py2exe、PyPy、PyInstaller 之类的工具可以将 Python 原始码转换成可以脱离 Python 解释器执行的程式。

15.6.1 官方镜像 library/python

使用 docker pull python 拉取 Python 镜像。Python 与 Node.js 在标签设置上非常相似，本节不再赘述，用户选择合适的标签拉取使用即可。

构建一个单文件的 Python 应用可以直接基于 CMD 构建。

```
FROM python:3-onbuild
# FROM python:2-onbuild
CMD [ "python", "./your-daemon-or-script.py" ]
```

当然更便捷的方式是使用 docker run 来运行：

```
$ docker run -it --rm \
  --name my-running-script \
  -v "$PWD":/usr/src/myapp \
  -w /usr/src/myapp \
  python:3 python your-daemon-or-script.py
```

需要提示的是 onbuild 标签的 ONBUILD 指令，以其中一个 onbuild 标签的 Dockerfile 为例：

```
FROM python:2.7

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY requirements.txt /usr/src/app/
ONBUILD RUN pip install --no-cache-dir -r requirements.txt

ONBUILD COPY . /usr/src/app
```

如上所示，工作目录是 /usr/src/app，因此数据卷可以选择该目录，此外 requirements.txt 是一个 Python 项目不可缺少的一部分。

15.6.2 Flask 框架

在 Compose 实战中已经体验过 Django 的部署，现在来看如何构建部署一个 Flask 镜像。

Flask 是一个使用 Python 编写的轻量级 Web 应用框架。基于 Werkzeug WSGI 工具箱和 Jinja2 模板引擎。Flask 使用 BSD 授权。Flask 依赖两个外部库，即 Jinja2 模板引擎和 Werkzeug WSGI 工具集。

有了这些信息后就可以构建 Flask 镜像了，Dockerfile 文件如下：

```
FROM alpine:edge

# Flask 基本环境
RUN apk add --no-cache bash git nginx uwsgi uwsgi-python py-pip \
    && pip install --upgrade pip \
    && pip install flask

# 设置应用文件夹
ENV APP_DIR /app

# 创建应用文件夹
RUN mkdir ${APP_DIR} \
    && chown -R nginx:nginx ${APP_DIR} \
    && chmod 777 /run/ -R \
    && chmod 777 /root/ -R

VOLUME [${APP_DIR}]
WORKDIR ${APP_DIR}

EXPOSE 80

COPY nginx.conf /etc/nginx/nginx.conf
COPY app.ini /app.ini
COPY entrypoint.sh /entrypoint.sh

# execute start up script
ENTRYPOINT ["/entrypoint.sh"]
```

其中，Nginx 的配置文件已经介绍过，不再赘述。

app.ini 文件内容如下：

```
[uwsgi]
plugins = /usr/lib/uwsgi/python
chdir = /app
module = app:app
uid = nginx
gid = nginx
socket = /run/uwsgiApp.sock
pidfile = /run/.pid
processes = 4
threads = 2
然后在/entrypoint.sh 里面只是让应用启动的两句命令
#!/bin/bash
pip install -r requirements.txt
nginx && uwsgi --ini /app.ini
```

完成上述工作，就可以构建整个镜像了。

```
$ docker build -t user/flask
```

运行时设定数据卷与端口即可访问。

```
$ docker run -d -v /path/app:/app -p 80:80 user/flask
```

15.6.3 基于 Alpine 构建 Python 镜像

Python 作为一门流行的编程语言并且容易移植，Alpine 必定早有其软件包，因此构建一个基于 Alpine 的 Python 镜像并非难事。

本节以 Python 2.7 和 Python 3 作为例子，先是 Python 2.7 版本的镜像构建。

```
FROM alpine:edge
RUN apk update && apk upgrade \
    && apk add ca-certificates python \
    && rm -rf /var/cache/apk/*
```

非常简单的 Dockerfile，默认 python 包就是 Python 2.7 版本的。如果需要开发版的 Python，可以这样构建：

```
FROM alpine:edge

RUN apk update && apk upgrade \
    && apk add curl ca-certificates python-dev \
    && curl -sS https://bootstrap.pypa.io/get-pip.py | python \
    && apk del curl && rm -rf /var/cache/apk/*
```

上面的 Dockerfile 安装了 python-dev 和 pip 工具，这是 Python 开发中最常用到的工具。再来看 Python 3 的构建，依旧是改个名字。

```
FROM alpine:edge
RUN apk update && apk upgrade \
    && apk add ca-certificates python3 \
    && rm -rf /var/cache/apk/*
```

使用开发版的 Python 3 和上面 Python 2.7 一样：

```
FROM alpine:edge
RUN apk update && apk upgrade \
    && apk add ca-certificates curl python3-dev \
    && curl -sS https://bootstrap.pypa.io/get-pip.py | python3 \
    && apk del curl && rm -rf /var/cache/apk/*
```

以上镜像全部使用 docker build 命令构建即可。

15.7 Swift 语言

Swift 语言是苹果于 2014 年 WWDC（苹果开发者大会）发布的新开发语言，Swift 是一款易学易用的编程语言，而且还是第一套具有与脚本语言同样的表现力和趣味性的系统编程语言。Swift 的设计以安全为出发点，以避免各种常见的编程错误类别。

2015 年 12 月 4 日，苹果公司宣布其 Swift 编程语言开放源代码。本书对 Swift 的介绍是基于 Docker 平台的，因此只会介绍 Swift for Linux。

15.7.1 构建 Swift 镜像

目前 Docker Hub 上并没有官方维护的 Swift 镜像，因此本节就不能介绍官方镜像了，

直接自己动手构建一个 Swift 镜像吧。

通过 Swift 的文档可以得知, Swift 的依赖有 curl、python-dev、libedit2、clang、libicu-dev、libxml2, 所以我们选择 ubuntu:trusty 作为基础镜像, 便于安装依赖:

```
FROM ubuntu:trusty
MAINTAINER ZuoLan <i@zuolan.me>

# 如果使用 Swift package, 请把 libicu52 替换为 libicu-dev
RUN swiftDeps="curl python-dev libedit2 clang libicu52 libxml2" \
    && apt-get update && apt-get -y install $swiftDeps \
    && cd /usr/local/ \
    && curl -o swift.tar.gz -sL https://swift.org/builds/swift-3.0-release/ \
    ubuntu1404/swift-3.0-RELEASE/swift-3.0-RELEASE-ubuntu14.04.tar.gz \
    && tar xzf swift.tar.gz && mv swift-3.0-RELEASE-ubuntu14.04 swift && rm \
    /usr/local/swift.tar.gz \
    && echo 'export PATH=/usr/local/swift/usr/bin:"${PATH}"' >> ~/.bashrc \
    && apt-get autoremove -y $buildDeps \
    && apt-get autoremove -y && apt-get autoclean -y && apt-get clean -y \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

CMD ["swift", "--version"]
```

构建镜像:

```
$ docker build -t user/swift .
```

因为 Swift 需要使用 LLVM, 因此启动时需要添加一个特别的参数--privileged, 给容器足够的权限。

```
$ docker run --rm --privileged user/swift
Swift version 3.0 (swift-3.0-RELEASE)
Target: x86_64-unknown-linux-gnu
```

15.7.2 Kitura 框架

Swift 在 Web 领域虽然刚刚起步, 但是 Swift 语言的设计特点吸引了不少用户, 因此有不少 Swift 的 Web 框架出现。

本节介绍的 Kitura 就是来自 IBM 公司的 Swift Web 框架。IBM 公司为 Kitura 构建了一个 Docker 镜像。Dockerfile 文件如下:

```
FROM ibmcom/swift-ubuntu:latest
MAINTAINER IBM Swift Engineering at IBM Cloud
LABEL Description="Docker image for building and running the Kitura-Starter \
sample application."

# Expose default port for Kitura
EXPOSE 8090

# Add utility build files to image
ADD clone_build_kitura.sh /root
ADD start_kitura_sample.sh /root

# Clone and build Kitura and sample app using utility script
RUN /root/clone_build_kitura.sh

USER root
CMD /root/start_kitura_sample.sh
```

构建过程并不复杂，基础镜像选择了 IBM 自己构建的 Swift 镜像，而且关键构建过程没有显示在 Dockerfile 中，而是使用 clone_build_kitura.sh 脚本来执行，我们来看这个脚本都做了什么。

```
set -e

REPO_URL=https://github.com/IBM-Bluemix/Kitura-Starter.git

# Clone and build Kitura-Starter
# The Git branch to clone should be passed as a parameter
# If not provided as a parameter, then using develop as the default value.
if [ -z "$1" ]; then
    KITURA_BRANCH="master"
else
    KITURA_BRANCH=$1
fi

echo ">> About to clone branch '$KITURA_BRANCH' for Kitura-Starter"
# Clone Kitura repo
cd /root && rm -rf Kitura-Starter && git clone -b $KITURA_BRANCH $REPO_URL

# Make the Kitura folder the working directory
cd /root/Kitura-Starter

# Build Kitura-Starter
echo ">> About to build Kitura-Starter..."
make
echo ">> Build for Kitura-Starter completed (see above for results)."
```

可以看到，实际上脚本也是复制一个叫做 Kitura-Starter 的项目，然后执行 make 操作。现在来尝试启动这个项目：

```
$ docker run -p 8090:8090 ibmcom/kitura-ubuntu
Status: Downloaded newer image for ibmcom/kitura-ubuntu:latest
INFO: Kitura_Starter main.swift line 29 - Server will be started on
'http://localhost:8090'.
INFO: listen(socket:port:) HTTPServer.swift line 128 - Listening on port
8090
```

打开浏览器 <http://localhost:8090> 可以看到如图 15.18 所示的欢迎页面。

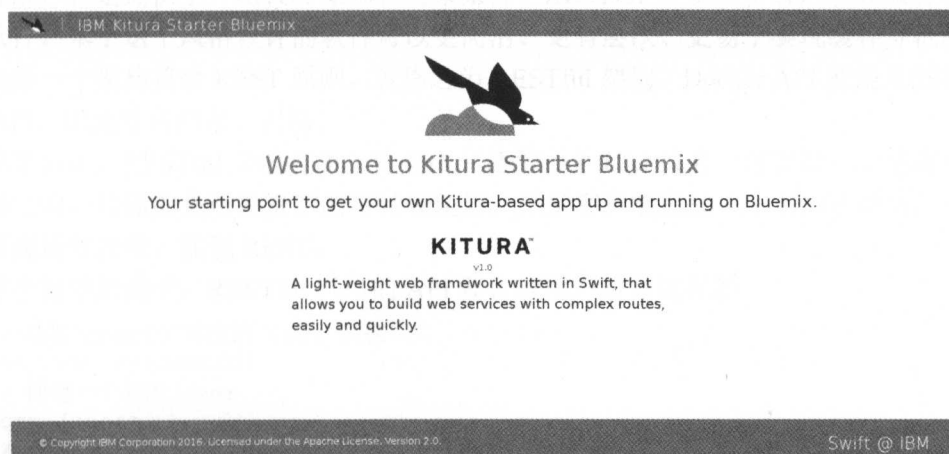


图 15.18 使用 Docker 部署 Kitura

有意思的是，笔者在写本节内容的时候，Kitura 镜像刚刚发布不到 20 小时，因此更多信息有待读者去发现。除了 Kitura，Swift 还有很多出色的 Web 框架，例如 Vapor、SwiftExpress、SwiftOn、Taylor 等。

15.8 本章小结

本章主要介绍了主流的几大编程语言，以及它们的框架或者应用在 Docker 上的应用。本章每节都是从认识官方镜像开始然后实战该语言的框架与应用，最后创建自己的定制镜像。

其实总的来说，每种编程语言在 Docker 化的过程中只要解决运行时的依赖问题，基本不会有其他问题。因此在 Docker 上构建部署应用的关键在于如何让镜像变得结构清晰、容易管理、轻松更新维护，如何在启动容器时考虑到不同的应用场景以及数据的存储管理操作是否便捷。这些经验不是仅凭一章的内容就可以解释完并领悟的，还需要读者在应用中多留心与思考，例如配合 shell 的特点构建出“精彩”的镜像。

关于文章的源代码除了特别指出外，其他均可在 <https://github.com/izuolan/dockerfiles> 中获取。

第 16 章 Docker API 介绍

Docker 能够快速成为当今最火热的容器项目与其优秀的 API 有着不可分割的关系，Docker API 的良好设计使得大量第三方工具迅速发展起来，形成围绕 Docker 的强大生态系统。

读者将通过本章学习 Docker API 来深入理解 Docker，本章提供的一些实战例子会让读者更容易理解如何把 Docker 集成到自己的系统中。

本章的主要知识点有：

- 认识 RESTful API。
- 掌握 Docker Remote API 的使用。
- 认识 Trusted Registry API 的使用。

16.1 认识 Docker API

Docker API 的设计十分出色离不开 REST (Representational State Transfer) 架构，那么什么是 REST？怎样可以称之为 RESTful API？本节将告诉答案。

16.1.1 RESTful 介绍

说起 Docker API，就不得不提 RESTful 这种架构，RESTful 是一种软件风格架构，而不是标准，只是提供了一组设计原则和约束条件。RESTful 主要用于客户端和服务端交互的软件。基于这个风格设计的软件可以更简洁、更有层次、更易于实现缓存等机制。

如果一个架构符合 REST 原则，就称之为 RESTful 架构。Docker API 就是 RESTful 风格的 API，因此非常清晰、灵活。

简单来说，RESTful 架构有 3 个特点：一是每一个 URI 代表一种资源；二是客户端和服务端之间，传递的是这种资源的某种表现层；三是客户端通过 4 个 HTTP 动词，对服务端资源进行操作，实现 REST。

举个简单的例子，RESTful 风格的 API 要管理用户时是这样的：

```
// 获取 userId 对应的 user 信息
get /users/{userId}
// 创建一个新的 user
post /users
// 更改 userId 对应的 user 信息
put /users/{userId}
// 删除 userId 对应的 user
delete /users/{userId}
```

具体关于 RESTful 的知识不在本节讨论范围，详细资料在网络上可以找到不少。

16.1.2 开启 socket

为了更好地学习 Docker API，首先要设置一下本地的 Docker daemon 启动参数，使其开启 socket 端口用于返回信息。

根据不同的发行版（这里以 Linux 为例，Windows 与 Mac OS 没有这个问题），开启 socket 的方式也有不同。简单来说，使用 service 命令管理服务的发行版（例如 Ubuntu 14.04）可以直接通过修改/etc/default/docker 文件就可以开启 socket：

```
# Docker Upstart and SysVinit configuration file

#
# THIS FILE DOES NOT APPLY TO SYSTEMD
#
# Please see the documentation for "systemd drop-ins":
# https://docs.docker.com/engine/articles/systemd/
#

# Customize location of Docker binary (especially for development testing).
#DOCKERD="/usr/local/bin/dockerd"

# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
# 修改这里
DOCKER_OPTS="-H tcp://127.0.0.1:2376 -H unix:///var/run/docker.sock"
# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"
export http_PROXY="socks5://127.0.0.1:1080"
export https_PROXY="socks5://127.0.0.1:1080"
# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

然后保存并重启 Docker：

```
$ service docker restart
```

现在可以使用 curl 等工具访问本地 Docker daemon 了。

至于使用 systemd 管理系统服务的发行版（例如 Ubuntu 16.04），需要在 /etc/systemd/system/ 文件夹中操作，具体如下。

创建文件夹：

```
$ sudo mkdir /etc/systemd/system/docker.service.d/
```

新建配置文件，内容如下：

```
$ sudo vim /etc/systemd/system/docker.service.d/remote-api.conf
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H tcp://127.0.0.1:2376 -H unix:///var/run/docker.sock
```

注意，ExecStart 要输入两次。保存后重新加载 systemd 配置并重启 Docker。

```
$ systemctl daemon-reload
$ systemctl restart docker
```

一般来说，设置 127.0.0.1 的 socket 时外界是无法访问的，因此不会有安全问题，但是

如果需要向外界开放 socket 或者远程管理 Daemon 时, socket 端口暴露的问题就不得不解决了, 毕竟直接使用上面的方法会造成 Docker daemon 的 socket 端口暴露, 因而有安全问题, 为了保护这个 socket 端口, 通常需要设置加密与认证, 详细信息请阅读官方文档 <https://docs.docker.com/engine/security/https/#connecting-to-the-secure-docker-port-using-curl>。

16.1.3 使用 curl

curl 是 Linux 下非常著名的一个工具, 本章中会大量使用到该工具, 如果用户的发行版没有安装该工具, 使用相应的包管理工具安装即可。

先来看一个简单的 Docker API 例子, 以 JSON 数据格式返回一个镜像的信息。

```
$ curl --unix-socket /var/run/docker.sock tcp::/images/nginx:alpine/json
{"Id":"sha256:935bd7bf8ea67cda09fd76cb2b3c8ac36e365fd68f30084da395fb0eb4e3200a","RepoTags":["nginx:alpine"],"RepoDigests":["nginx@sha256:01a97a837e2af35ae0e2bf0d5609af4b1e2f698f0856b36e3975c9d5eec3f836"],"Parent":"","Comment":"","Created":"2016-08-23T18:53:03.518921298Z","Container":"64fdc629057a22025376453b2d4ea002df1657528ace0f2663675d5b7111b777","ContainerConfig":{"Hostname":"4c573a7cf4a3","Domainname":"","User":"","AttachStdin":false,"AttachStdout":false,"AttachStderr":false,"ExposedPorts":{"443/tcp":{},"80/tcp":{},"Tty":false,"OpenStdin":false,"StdinOnce":false,"Env":["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin","NGINX_VERSION=1.11.3"],"Cmd":["/bin/sh","-c","#(nop) CMD [\"nginx\\\"\\\"-g\"\\\" daemon off;\\\"]"],"ArgsEscaped":true,"Image":"sha256:267edbb122b5bfa5842a4cb80e4b254e70b10a9b6f1e9ab7713ad64e6f03cbde","Volumes":null,"WorkingDir":"","Entrypoint":null,"OnBuild":[],"Labels":{},"DockerVersion":"1.10.3","Author":"NGINX Docker Maintainers <docker-maint@nginx.com>","Config":{"Hostname":"4c573a7cf4a3","Domainname":"","User":"","AttachStdin":false,"AttachStdout":false,"AttachStderr":false,"ExposedPorts":{"443/tcp":{},"80/tcp":{},"Tty":false,"OpenStdin":false,"StdinOnce":false,"Env":["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin","NGINX_VERSION=1.11.3"],"Cmd":["nginx","-g","daemon off;"],"ArgsEscaped":true,"Image":"sha256:267edbb122b5bfa5842a4cb80e4b254e70b10a9b6f1e9ab7713ad64e6f03cbde","Volumes":null,"WorkingDir":"","Entrypoint":null,"OnBuild":[],"Labels":{},"Architecture":"amd64","Os":"linux","Size":54810234,"VirtualSize":54810234,"GraphDriver":{"Name":"devicemapper","Data":{"DeviceId":"137","DeviceName":"docker-8:1-1059258-e3e3e1cc572b31897ac91e7b237c5dcb83f55c6b1fca8049098891234a5a2864","DeviceSize":"10737418240"}}, "RootFS":{"Type":"layers","Layers":["sha256:4fe15f8d0ae69e169824f25f1d4da3015a48feeeebb265cd2e328e15c6a869f","sha256:da85d9b3377006fe88628078bd0b0418b81fc9d4018f903169a716214479ee64","sha256:b36519590516cd2cfda9f00c20ce06d64f0375e1ad895cd26827a2dbdb9e2e1d","sha256:1bbf5c4f940bae3e55efe6bbf4433361f5101727f7440755f2c4e87a6b967297"]}}}}
```

返回的信息是不是非常熟悉? 没错, 与 docker inspect 命令返回的信息非常相似。当然 docker inspect 返回的信息是经过格式化的。

如果没有返回信息或者返回错误信息, 可以给 curl 命令添加 -v 参数, 这样会输出更详细的信息, 以便分析失败原因。

具体如何使用 curl 命令进行 Docker API 的操作在后面会详细讲解, 本节只是让读者体

验到 Docker API 的灵活、方便。

上面的 curl 命令中，`--unix-socket /var/run/docker.sock` 表示从指定的文件中读取信息，然后 `tcp::images/nginx:alpine/json` 很明显是一种格式，例如，查看 Ubuntu 镜像信息就是 `tcp::images/ubuntu/json`，即便是第一次看也可以知道这表示什么意思，这就是 RESTful 的一个好处，API 的结构十分清晰。

16.1.4 使用 Postman

前面使用终端返回那么多密密麻麻的信息不容易阅读，虽然可以使用一些其他工具（例如 `python -mjson.tool`）来辅助格式化 JSON 信息，但是，笔者更推荐使用一个更易用的工具 Postman，它是一个 Google Chrome 浏览器的插件，从商店安装即可。FireFox 下也有一个 RESTClient 的插件。Postman 返回的信息更加直观，如图 16.1 所示，因此在后面也会用到它。

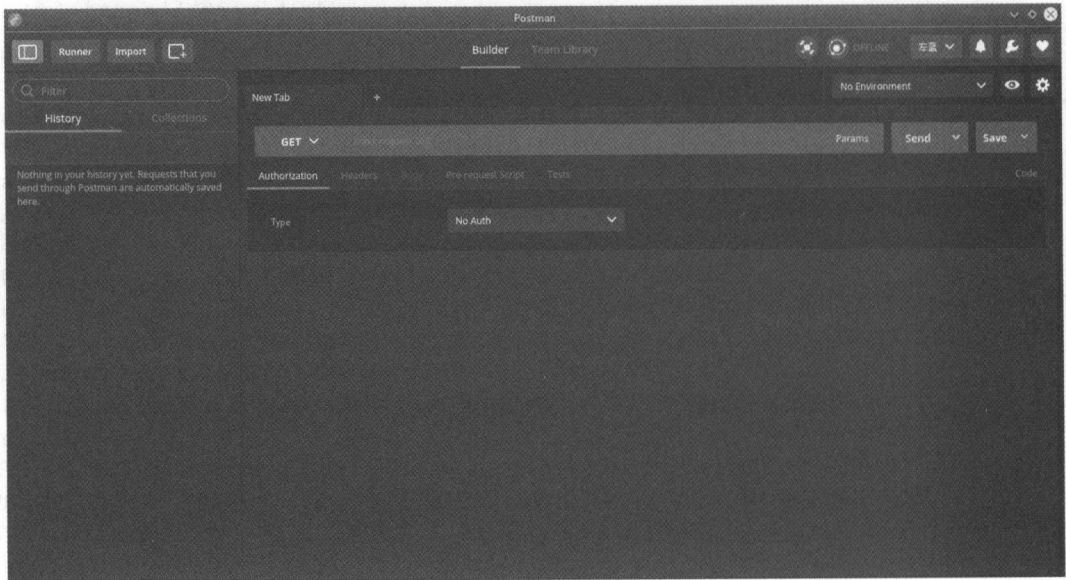


图 16.1 Postman 界面

在 Postman 的左边可以看到历史信息，还可以创建收藏命令，非常方便。现在使用 Postman 来获取本地全部镜像的信息：

如图 16.2 所示，在请求 URL 地址的地方输入本机 Daemon 的 socket 地址。



图 16.2 请求 URL

单击 Send 按钮发送请求，就可以在下面看到本地的全部镜像信息了。

如图 16.3 所示，在左边可以看到刚才输入的请求历史，窗体下方返回的信息经过了格式化，用户可以根据自己的需要调整内容的格式。

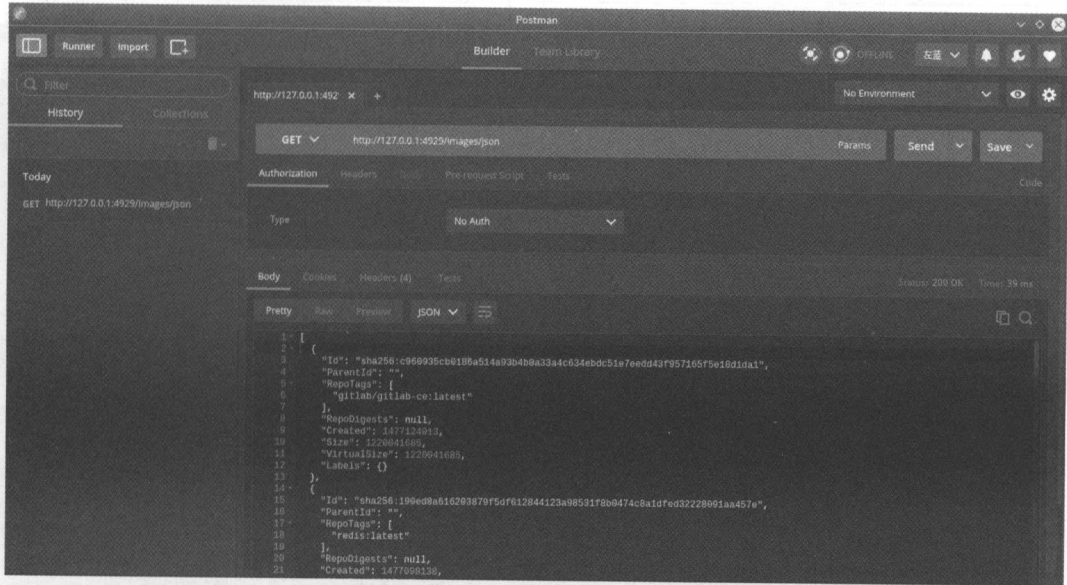


图 16.3 Postman 获取本地镜像信息

例如，查看上面 curl 请求的 `nginx:alpine` 镜像信息，可以输入相应的 socket 地址：

`http://127.0.0.1:4929/images/nginx:alpine/json`

返回的信息如下，经过了格式化。

```

{
  "Id": "sha256:935bd7bf8ea67cda09fd76cb2b3c8ac36e365fd68f30084da395fb0eb
4e3200a",
  "RepoTags": [
    "nginx:alpine"
  ],
  "RepoDigests": [
    "nginx@sha256:01a97a837e2af35ae0e2bf0d5609af4b1e2f698f0856b36e3975c9
d5eec3f836"
  ],
  "Parent": "",
  "Comment": "",
  "Created": "2016-08-23T18:53:03.518921298Z",
  "Container": "64fdc629057a22025376453b2d4ea002df1657528ace0f2663675d5b
7111b777",
  ... ..
  "Architecture": "amd64",
  "Os": "linux",
  "Size": 54810234,
  "VirtualSize": 54810234,
  "GraphDriver": {
    "Name": "devicemapper",
    "Data": {
      "DeviceId": "137",
      "DeviceName": "docker-8:1-1059258-e3e3e1cc572b31897ac91e7b237c5dcb8
3f55c6b1fca8049098891234a5a2864",
      "DeviceSize": "10737418240"
    }
  },
  "RootFS": {
    "Type": "layers",
    "Layers": [

```

```

    "sha256:4fe15f8d0ae69e169824f25f1d4da3015a48feeeeebb265cd2e328e15c
    6a869f",
    "sha256:da85d9b3377006fe88628078bd0b0418b81fc9d4018f903169a7162144
    79ee64",
    "sha256:b36519590516cd2cfda9f00c20ce06d64f0375e1ad895cd26827a2dbdb
    9e2e1d",
    "sha256:1bbf5c4f940bae3e55efe6bbf4433361f5101727f7440755f2c4e87a6b
    967297"
  ]
}
}

```

相信读者已经发现，其实直接访问 <http://127.0.0.1:4929/images/nginx:alpine/json> 这个地址，也可以获得相应的信息，这体现了 Docker API 灵活的一方面。

16.2 Docker Remote API 介绍

目前 Docker 提供了多类 RESTful API，包括本章主要介绍的 3 类 API，即 Docker Registry API、Docker Hub API、Docker Remote API。除了这 3 类，本章也会介绍 Docker OAuth API、Swarm API 等。

Docker Remote API 是一个取代远程命令行界面 (rccli) 的 REST API。简单来说，Docker Remote API 指的是如 `docker run` 的这种操作，通常是 Docker client 发送给 Docker daemon 的请求。

16.2.1 容器 API

1. 容器列表

获取所有容器的信息可以使用 API：

```
GET /containers/json
```

例如，使用 `curl` 配合格式化工具 `python -mjson.tool`，可以获得输出信息如下（信息太多，这里只给出一行）：

```
$ curl --unix-socket /var/run/docker.sock tcp::/containers/json | python
-mjson.tool
```

在输出信息中可以看到，其实 API 与 `docker inspect` 返回的信息在细节上是有不小差别的。例如，在 API 信息中的 `Created` 的值与 `docker inspect` 返回的值表述是不一样的，像下面这样：

```
"Created": 1474796020,
"Created": "2016-10-27T01:27:24.529958691Z"
```

但实际上它们的值是相同的，前者表示从 1970 年 1 月 1 日至今流逝的秒数，后者使用的是我们能快速理解的日期表示方式。

除了使用 `curl` 获取 API，还可以使用 Postman 来获取，如图 16.4 所示。

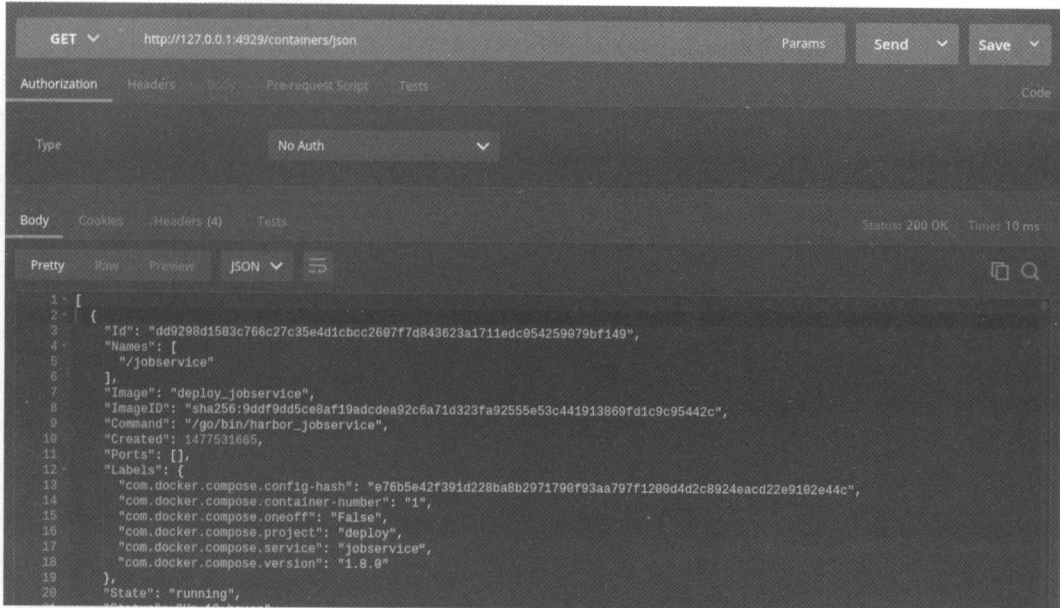


图 16.4 Postman 获取容器列表

Postman 可以保存输入历史，因此使用起来比较方便，还可以自动格式化返回信息。

2. 创建容器

创建新容器的 API 如下：

POST /containers/create

在 Docker API 中没有 docker run 这样的概念，前面使用的 docker run 实际上是 create 加上 start 的结果，所以 Docker API 中是没有 POST /containers/run 这样的 API 的。下面以创建一个容器为例，注意只是创建，并非运行。

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:4292/containers/create -d '{
```

```
  "Hostname": "",
  "Domainname": "",
  "User": "",
  "AttachStdin": false,
  "AttachStdout": true,
  "AttachStderr": true,
  "Tty": false,
  "OpenStdin": false,
  "StdinOnce": false,
  "Env": [
    "FOO=bar",
    "BAZ=quux"
  ],
  "Cmd": [
    "date"
  ],
  "Entrypoint": "",
  "Image": "ubuntu",
  "Labels": {
    "com.example.vendor": "Acme",
```

```

        "com.example.license": "GPL",
        "com.example.version": "1.0"
    },
    "Volumes": {
        "/volumes/data": {}
    },
    "WorkingDir": "",
    "NetworkDisabled": false,
    "MacAddress": "12:34:56:78:9a:bc",
    "ExposedPorts": {
        "22/tcp": {}
    },
    "StopSignal": "SIGTERM",
    "HostConfig": {
        "Binds": ["/tmp:/tmp"],
        "Links": ["redis3:redis"],
        "Memory": 0,
        "MemorySwap": 0,
        "MemoryReservation": 0,
        "KernelMemory": 0,
        "CpuPercent": 80,
        "CpuShares": 512,
        "CpuPeriod": 100000,
        "CpuQuota": 50000,
        "CpusetCpus": "0,1",
        "CpusetMems": "0,1",
        "IOMaximumBandwidth": 0,
        "IOMaximumIOps": 0,
        "BlkioWeight": 300,
        "BlkioWeightDevice": [{}],
        "BlkioDeviceReadBps": [{}],
        "BlkioDeviceReadIOps": [{}],
        "BlkioDeviceWriteBps": [{}],
        "BlkioDeviceWriteIOps": [{}],
        "MemorySwappiness": 60,
        "OomKillDisable": false,
        "OomScoreAdj": 500,
        "PidMode": "",
        "PidsLimit": -1,
        "PortBindings": { "22/tcp": [{ "HostPort": "11022" }] },
        "PublishAllPorts": false,
        "Privileged": false,
        "ReadonlyRootfs": false,
        "Dns": ["8.8.8.8"],
        "DnsOptions": [""],
        "DnsSearch": [""],
        "ExtraHosts": null,
        "VolumesFrom": ["parent", "other:ro"],
        "CapAdd": ["NET_ADMIN"],
        "CapDrop": ["MKNOD"],
        "GroupAdd": ["newgroup"],
        "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
        "NetworkMode": "bridge",
        "Devices": [],
        "Sysctls": { "net.ipv4.ip_forward": "1" },
        "Ulimits": [{}],
        "LogConfig": { "Type": "json-file", "Config": {} },
        "SecurityOpt": [],
        "StorageOpt": {},
        "CgroupParent": "",
        "VolumeDriver": ""
    }
}

```

```

    "ShmSize": 67108864
  },
  "NetworkingConfig": {
    "EndpointsConfig": {
      "isolated_nw": {
        "IPAMConfig": {
          "IPv4Address": "172.20.30.33",
          "IPv6Address": "2001:db8:abcd::3033",
          "LinkLocalIPs": ["169.254.34.68", "fe80::3468"]
        },
        "Links": ["container_1", "container_2"],
        "Aliases": ["server_x", "server_y"]
      }
    }
  }
}

```

如果创建成功会返回一个创建成功信息:

```

{"Id":"ca00cb670975d960a0d2bf00fe82d85ce30b5656b083920e4f0dcf5fa6738f25", "Warnings":null}

```

在上面的例子中几乎演示了全部的启动参数, 分别对应了 `docker run` 的各项参数, 本节会对其中的几个参数做介绍, 其他参数可以在官方的 API 文档获得帮助。

其实上面的内容略微看一下就知道代表的是什么意思, 例如 `Hostname` 设置容器的 `hostname`, `User` 设置容器执行命令时的用户, `Tty` 设置是否提供 TTY, `Env` 实际上就是 `docker run -e` 的意思, 还有 `Cmd`、`Entrypoint`、`Image`、`Volumes` 等这些都很熟悉的名字, 不用解释也知道是什么意思了。

现在的问题是, 怎么知道创建容器的格式? 换句话说创建一个容器需要这么多参数吗? 当然不是, 实际上上面只是一个作为参考的例子, 在使用 API 创建容器时完全可以只输入用到的参数:

```

$ curl -X POST -H "Content-Type: application/json" http://localhost:4929/containers/create?name=test -d '
{
  "Hostname":"testhost",
  "Cmd":["/bin/sh","-c","cat /etc/hosts"],
  "Image":"nginx:alpine"
}
'

```

就像上面这样也可以创建一个容器。

`create` 的 API 只有一个参数, 那就是 `name`, 就像上面的例子, 该参数可以分配容器的指定名称。格式如下:

```
/?[a-zA-Z0-9_-]+
```

例如例子中:

```
containers/create?name=test
```

容器的名字就是 `test`。在后面会启动这个容器。

3. 监控容器

使用容器 ID 获取该容器底层信息的 API:

```
GET /containers/(id)/json
```


使用 `curl` 来获取容器信息，这里的(id)可以是容器的 ID，也可以是容器名称：

```
$ curl --unix-socket /var/run/docker.sock tcp::/containers/nginx/json |
python -mjson.tool
.....
{
  "AppArmorProfile": "",
  "Args": [
    "-g",
    "daemon off;"
  ],
  .....
```

上面以本地一个名为 `nginx` 的容器为例，获得该容器的信息。

同样的，使用 `Postman` 也可以获得该信息，如图 16.5 所示。

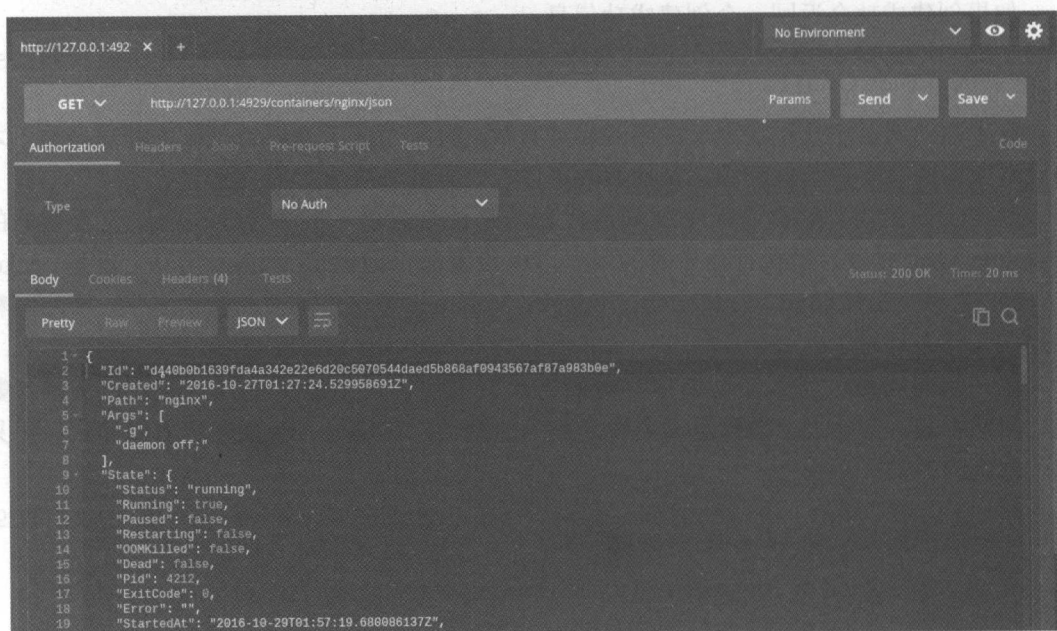


图 16.5 Postman 获取容器信息

4. 进程列表

获取容器内进程清单的 API:

GET /containers/(id)/top

这也是 `Docker` 容器的 API，使用 `curl` 来获取：

```
$ curl --unix-socket /var/run/docker.sock tcp::/containers/nginx/top
{"Processes":
  [{"root", "4212", "4143", "0", "10 月 29", "?", "00:00:00", "nginx: master process
nginx -g daemon off;"},
  ["systemd+", "4693", "4212", "0", "10 月 29", "?", "00:00:03", "nginx: worker
process"],
  ["systemd+", "4694", "4212", "0", "10 月 29", "?", "00:00:03", "nginx: worker
process"]},
```

```
[
  ["systemd+", "4695", "4212", "0", "10 月 29", "?", "00:00:01", "nginx: worker process"],
  ["systemd+", "4696", "4212", "0", "10 月 29", "?", "00:00:01", "nginx: worker process"]],
  "Titles": ["UID", "PID", "PPID", "C", "STIME", "TTY", "TIME", "CMD"]
}
```

也可以使用 Postman 来获取，如图 16.6 所示。

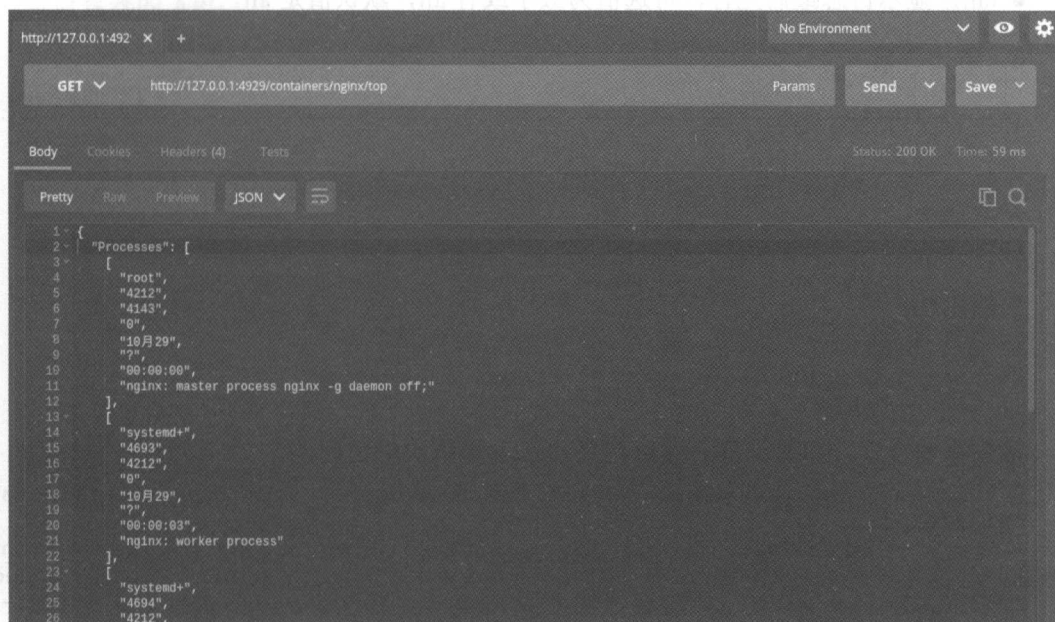


图 16.6 Postman 获取容器进程

这个 API 有一个可选参数 `ps_args`，类似 Linux 下 `ps` 命令的参数，就像 `aux` 一样，默认是 `-ef`。

5. 容器日志

获取容器的标准输出和错误日志的 API:

GET /containers/(id)/logs

使用示例 (curl) 如下:

```
$ curl --unix-socket /var/run/docker.sock tcp://containers/nginx/logs?
stderr=1&stdout=1&timestamps=1&follow=1&tail=10&since=1428990821
```

这个 API 只适用 `json-file` 或者 `journald` 这两种日志驱动，默认容器日志驱动是 `json-file`。

返回信息示例如下:

```
HTTP/1.1 101 UPGRADED
Content-Type: application/vnd.docker.raw-stream
Connection: Upgrade
Upgrade: tcp
{{ STREAM }}
```

这个 API 拥有不少参数，说明如下。

- `details`: 显示额外的详细信息提供给日志，可选值有 `1/True/true` or `0/False/flase`，默认 `false`。

- **follow**: 保持输出, 可选值和默认值同 **details**。
- **stdout**: 显示 **stdout** 日志, 可选值和默认值同 **details**。
- **stderr**: 显示 **stderr** 日志, 可选值和默认值同 **details**。
- **since**: 显示从什么时候到现在的日志, 格式是时间戳。
- **timestamps**: 在日志中显示时间戳, 可选值有 **1/True/true** 或 **0/False/flase**, 默认 **false**。
- **tail**: 显示日志最后几行, 可选值为数字或者 **all**, 默认值是 **all**。

6. 导出容器

导出容器内容的 API:

```
GET /containers/(id)/export
```

示例请求:

```
GET /containers/4fa6e0f0c678/export HTTP/1.1
```

实例返回:

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
{{ TAR STREAM }}
```

输出会非常多, 容易卡屏。可以使用 **curl -o** 来解决输出问题。

```
$ curl -o nginx.tar --unix-socket /var/run/docker.sock tcp::containers/nginx/export
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload    Upload   Total   Spent    Left   Speed
100 53.1M    0 53.1M    0     0      48.8M  0 --:--:--  0:00:01 --:--:--
48.8M
```

这个 API 没有参数。

7. 启动容器

启动容器的 API:

```
POST /containers/(id)/start
```

使用前面创建的 **nginx** 为例:

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:4929/containers/test/start
$ docker logs test
127.0.0.1      localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.16.0.10   testhost
```

上面例子中 **test** 的 CMD 是 **cat /etc/hosts**, 所以在日志中可以看到 **hosts** 的信息。

8. 停止容器

停止容器的 API:

```
POST /containers/(id)/stop
```

示例如下:

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  http://localhost:4929/containers/test/stop?t=5
```

其中有一个参数 `t`，设置停止响应等待的秒数，过后将停止容器。

9. 重启容器

重启容器的 API:

```
POST /containers/(id)/restart
```

示例如下:

```
$ curl -X \
  POST -H "Content-Type: application/json" \
  http://localhost:4929/containers/test/restart?t=5
```

参数和上面 `stop` 的 API 一样。

10. 终止容器

终止容器的 API:

```
POST /containers/(id)/kill
```

示例如下:

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  http://localhost:4929/containers/test/kill
```

其中有一个参数 `signal`，作用是改变 `kill` 发送的信号。

Docker API 容器管理方面常用的 API 就是这些，还有一些未提及的 API 可以在官方文档 <https://docs.docker.com/engine/reference/api/> 中查看。

16.2.2 镜像 API

在介绍了容器的 API 之后，相信大部分读者都已经知道镜像 API 的结构了，本节依旧使用 `curl` 演示为主。

1. 创建镜像

创建镜像实际上是从 `Registry` 拉取或者导入镜像。API 是:

```
POST /images/create
```

拉取示例如下:

```
POST /images/create?fromImage=busybox&tag=latest HTTP/1.1
```

返回信息如下:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
{ "status": "Pulling..." }
{ "status": "Pulling", "progress": "1 B / 100 B", "progressDetail": { "current":
1, "total": 100 } }
{ "error": "Invalid..." }
.....
```

参数如下。

- fromImage: 表示镜像拉取名称, orensh 默认是从 Docker Hub 拉取。
- fromSrc: 指定导入的资源。
- repo: 导入镜像时指定镜像名称。
- tag: 指定拉取镜像时的标签, 默认是 latest。

2. 使用容器创建镜像

使用容器创建镜像的 API 是:

POST /commit

示例 (curl) 如下:

POST /commit?container=44c004db4b17&comment=message&repo=myrepo HTTP/1.1
Content-Type: application/json

```
{
  "Hostname": "",
  "Domainname": "",
  "User": "",
  "AttachStdin": false,
  "AttachStdout": true,
  "AttachStderr": true,
  "Tty": false,
  "OpenStdin": false,
  "StdinOnce": false,
  "Env": null,
  "Cmd": [
    "date"
  ],
  "Mounts": [
    {
      "Source": "/data",
      "Destination": "/data",
      "Mode": "ro,Z",
      "RW": false
    }
  ],
  "Labels": {
    "key1": "value1",
    "key2": "value2"
  },
  "WorkingDir": "",
  "NetworkDisabled": false,
  "ExposedPorts": {
    "22/tcp": {}
  }
}
```

示例返回信息如下:

HTTP/1.1 201 Created
Content-Type: application/json

```
{"Id": "596069db4bf5"}
```

参数 config 指定容器配置, 格式是 JSON; container 指定要 commit 的容器名称; repo 参数指定提交镜像的名称; tag 参数指定标签; comment 参数设置提交信息; author 参数输入作者信息; pause 参数设置是否要在提交前暂停容器, 可选值有 1/True/true or 0/False/false; changes 参数设置提交时修改 Dockerfile 指令。

3. 镜像列表

获取本地镜像列表的 API:

GET /images/json

示例 (curl) 如下:

```
$ curl --unix-socket /var/run/docker.sock tcp::images/json | python -mjson.tool
```

```
.....
[
  {
    "Created": 1477124013,
    "Id": "sha256:c960935cb0186a514a93b4b0a33a4c634ebdc51e7eedd43f957165f5e18d1da1",
    "Labels": {},
    "ParentId": "",
    "RepoDigests": null,
    "RepoTags": [
      "gitlab/gitlab-ce:latest"
    ],
    "Size": 1220041685,
    "VirtualSize": 1220041685
  },
  {
    "Created": 1477098138,
    "Id": "sha256:190ed8a616203879f5df612844123a98531f8b0474c8a1dfed32228091aa457e",
    "Labels": {},
    "ParentId": "",
    "RepoDigests": null,
    "RepoTags": [
      "redis:latest"
    ],
    "Size": 182837415,
    "VirtualSize": 182837415
  }
],
```

上面的 Size 表示的是该层相较于上一层新增的大小，VirtualSize 表示镜像整体大小。使用 Postman 阅读起来更轻松，如图 16.7 所示。

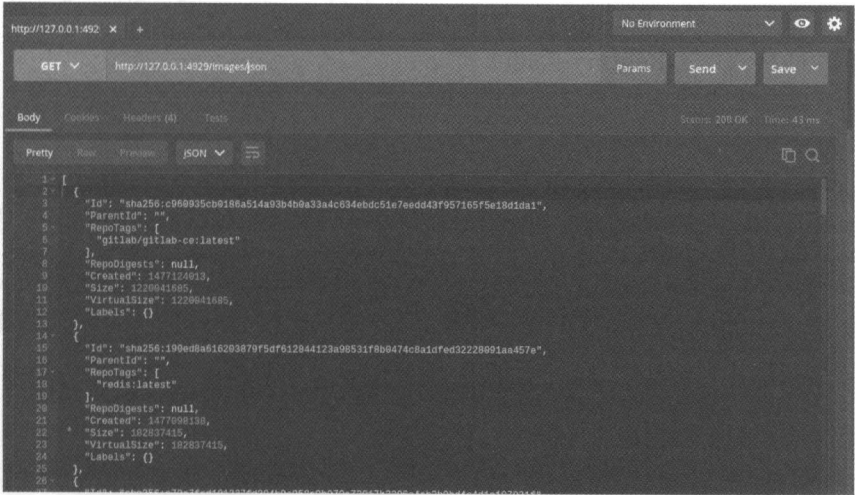


图 16.7 Postman 获取镜像列表

这个 API 的 3 个参数值与 docker images 的参数作用相同。

- all: 显示全部镜像, 包括 dangling 镜像, 默认 false。
- filters: 按照模板返回, 例如 dangling=true。
- filter: 只返回指定名称的镜像。

4. 删除镜像

删除镜像的 API:

DELETE /images/(name)

示例如下:

```
$ curl -X DELETE http://localhost:4929/images/redis
[{"Untagged":"redis:latest"}, {"Deleted":"sha256:190ed8a616203879f5df612844123a98531f8b0474c8aldfed32228091aa457e"}, {"Deleted":"sha256:74d1dfbdf8e9e4e33799d168a9b2d39e916bd0d6f8c118cd2181cb87420dc54b"}, {"Deleted":"sha256:2b1ea9bbe62549a88bff5bcac616b20668cb4916b454a445a4882a9e20ced5c7"}, {"Deleted":"sha256:96830f23feeb1123b7d3d1e74181962997b44a35c646ed32fe91117371305ea2"}, {"Deleted":"sha256:614250b95cd1ce5ce7f05191976898ff1fba6240840bdd1d6af6212fd53439a2"}, {"Deleted":"sha256:124c27df805e55420b70a2a3592ae5f706dca3f5aae62ef6dd61ab2d7ab0c4b2"}, {"Deleted":"sha256:10ab4d2da45175e27520a007402e84d05d34c63eb25a128af5e60c84cddb65d5"}, {"Deleted":"sha256:f96222d75c5563900bc4dd852179b720a0885de8f7a0619ba0ac76e92542bbc8"}]
```

其中有两个参数, 分别是 force 和 noprun, 前者表示强制删除, 后者默认值为 false。

5. 推送镜像

推送镜像的 API:

POST /images/(name)/push

为方便演示, 这里不加入验证环节, 实际应用中, 大部分仓库是有验证的, 验证的 API 在后面会讲。

示例如下:

```
POST /images/registry.acme.com:5000/test/push HTTP/1.1
```

有一个参数 tag, 可以向镜像打上标签。

验证仓库的格式 (X-Registry-Auth) 如下:

```
{ "username": "jdoe", "password": "secret", "email": "jdoe@acme.com", }
```

或者

```
{ "identitytoken": "9cbaf023786cd7..." }
```

认证内容加在 -H "... .." 当中。

```
$ curl -X POST -H "X-Registry-Auth { \"identitytoken\": \"9cbaf023786cd7...\" }"
http://localhost:4929/images/<镜像 ID 或者镜像名称>/push
```

6. 打上标签

给镜像打上标签的 API:

POST /images/(name)/tag

示例 (curl) 如下:

```
$ curl -X POST http://localhost:4929/images/nginx:alpine/tag\?repo\=test
```

参数有两个，分别是 `repo` 和 `tag`，前者指定镜像名称，后者指定标签。

7. 搜索镜像

搜索镜像的 API:

```
GET /images/search
```

默认是从 Docker Hub 上搜索。

示例如下:

```
GET /images/search?term=sshd HTTP/1.1
```

返回信息如下:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
[
  {
    "description": "",
    "is_official": false,
    "is_automated": false,
    "name": "wma55/ul210sshd",
    "star_count": 0
  },
  {
    "description": "",
    "is_official": false,
    "is_automated": false,
    "name": "jdswinbank/sshd",
    "star_count": 0
  },
  {
    "description": "",
    "is_official": false,
    "is_automated": false,
    "name": "vgauthier/sshd",
    "star_count": 0
  }
]
.....
]
```

参数有 3 个，分别是 `term`、`limit` 和 `filters`，`term` 指定镜像名称，`limit` 指定返回数据条数，`filters` 表示过滤条件。过滤条件有 3 个可选值，分别是 `stars=<number>`、`is-automated=(true|false)` 和 `is-official=(true|false)`。

8. 镜像历史

查看镜像历史的 API:

```
GET /images/(name)/history
```

示例 1 (Postman) 如图 16.8 所示。

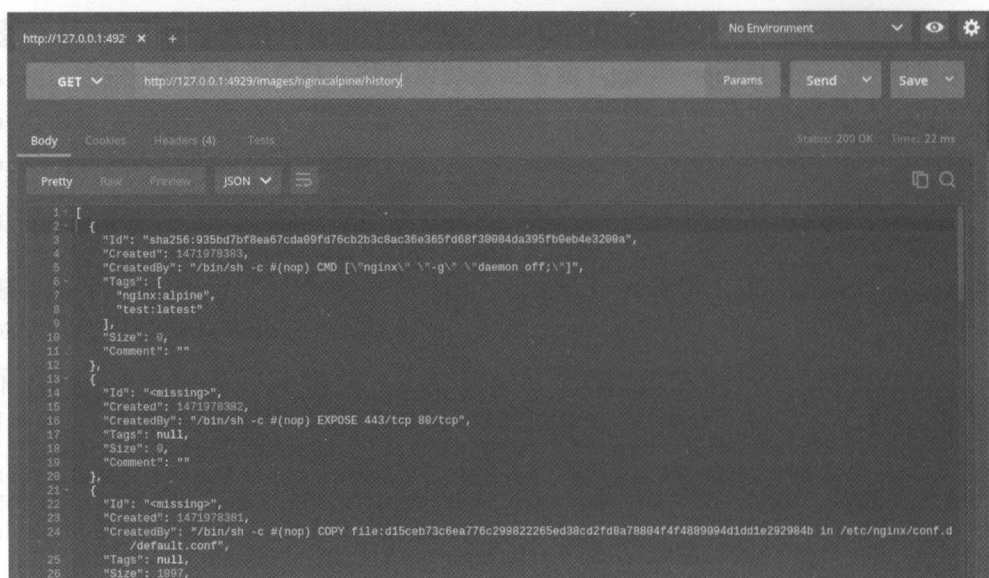


图 16.8 Postman 获取镜像历史

示例 2 (curl) 如下:

HTTP/1.1 200 OK

Content-Type: application/json

```
[
  {
    "Id": "3db9c44f45209632d6050b35958829c3a2aa256d81b9a7be45b362ff85c54710",
    "Created": 1398108230,
    "CreatedBy": "/bin/sh -c #(nop) ADD file:eb15dbd63394e063b805a3c32ca7bf0266ef64676d5a6fab4801f2e81e2a5148 in /",
    "Tags": [
      "ubuntu:lucid",
      "ubuntu:10.04"
    ],
    "Size": 182964289,
    "Comment": ""
  },
  {
    "Id": "6cfa4dlf33fb861d4d114f43b25abd0ac737509268065cdfd69d544a59c85ab8",
    "Created": 1398108222,
    "CreatedBy": "/bin/sh -c #(nop) MAINTAINER Tianon Gravi <admwiggin@gmail.com> - mkimage-debootstrap.sh -i iproute,iputils-ping,ubuntu-minimal -t lucid.tar.xz lucid http://archive.ubuntu.com/ubuntu/",
    "Tags": null,
    "Size": 0,
    "Comment": ""
  },
  {
    "Id": "511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158",
    "Created": 1371157430,
    "CreatedBy": "",
    "Tags": [
      "scratch12:latest",

```



```

    "scratch:latest"
  ],
  "Size": 0,
  "Comment": "Imported from -"
}
]

```

该 API 没有参数。

9. 构建镜像

构建镜像的 API:

POST /build

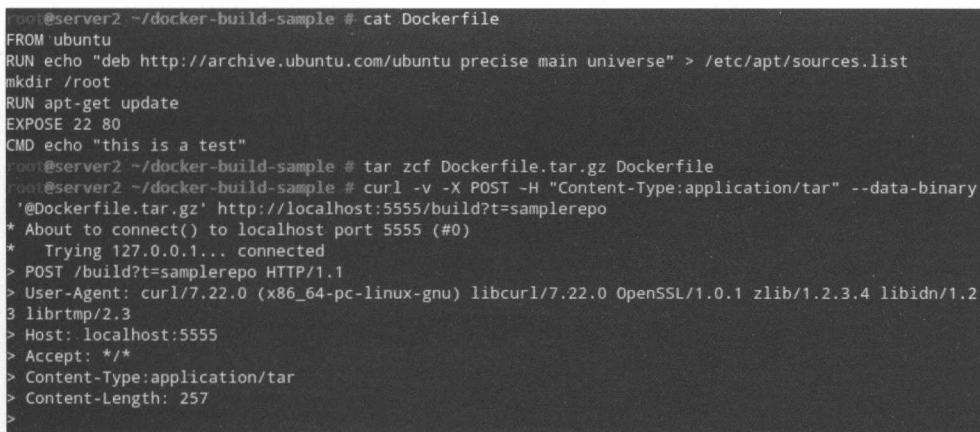
示例 (curl) 如下:

```

$ tar zcf Dockerfile.tar.gz Dockerfile
$ curl -X POST -H "Content-Type: application/tar" --data-binary '@Dockerfile.
tar.gz' http://localhost:4929/build?t=test_images

```

如图 16.9 所示, 这个 API 的参数非常多, 这里就不一一解释了, 因为前面第 4 章中已经详细介绍过一部分。



```

root@server2 ~/docker-build-sample # cat Dockerfile
FROM ubuntu
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
mkdir /root
RUN apt-get update
EXPOSE 22 80
CMD echo "this is a test"
root@server2 ~/docker-build-sample # tar zcf Dockerfile.tar.gz Dockerfile
root@server2 ~/docker-build-sample # curl -v -X POST -H "Content-Type:application/tar" --data-binary
 '@Dockerfile.tar.gz' http://localhost:5555/build?t=samlerepo
* About to connect() to localhost port 5555 (#0)
* Trying 127.0.0.1... connected
> POST /build?t=samlerepo HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.2
3 librtmp/2.3
> Host: localhost:5555
> Accept: */*
> Content-Type:application/tar
> Content-Length: 257
>

```

图 16.9 使用 API 构建镜像

以上就是常用的镜像 API, 其实还有不少 API 没有介绍, 但是细心地看, 其实 Docker API 的结构并不复杂, 很容易就能掌握。关键在于参数的设置上, 要依靠官方文档的资料。目前文档的版本是 1.24, 如果读者看到本书时版本已经更新, 则以官方文档为准。

16.3 其他 API

Docker 提供了很多 API 以方便用户使用。这些 API 包含 4 个方面: Docker Registry API、Docker Hub API、Docker OAuth API 和 Docker Remote API, 前面 16.1 节和 16.2 节介绍的都属于 Remote API。本节将介绍其他的部分 API, 更多 API 使用, 在官方文档有详细说明。

16.3.1 常用 API

1. 登录API

登录到 Docker Hub

POST /auth

示例登录:

POST /auth HTTP/1.1

Content-Type: application/json

```
{
  "username": "hannibal",
  "password": "xxxx",
  "serveraddress": "https://index.docker.io/v1/"
}
```

示例返回:

HTTP/1.1 200 OK

```
{
  "Status": "Login Succeeded",
  "IdentityToken": "9cbaf023786cd7..."
}
```

2. 获取Info API

获取 Docker info 信息的 API:

GET /info

示例 (curl) 如下:

\$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:4929/info

示例 (Postman) 如图 16.10 所示。

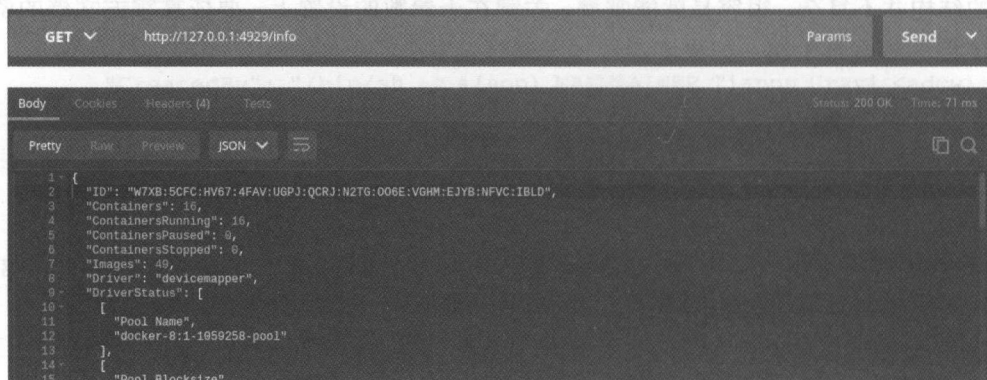


图 16.10 Postman 获取 Docker Info

3. 版本信息API

显示 Docker 版本信息，格式如下：

```
GET /version
```

使用示例 (curl) 如下：

```
$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:4929/version
```

4. Daemon状态

查看 Daemon 状态，格式如下：

```
GET /_ping
```

使用示例 (curl) 如下：

```
$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:4929/_ping
```

5. 事件API

查看 Docker 的事件，格式如下：

```
GET /events
```

使用示例 (curl) 如下：

```
$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:4929/events
```

6. 创建exec

创建 exec 可以使用户进入容器内部，格式如下：

```
POST /containers/(id or name)/exec
```

示例 (curl) 如下：

```
$ curl -X POST \
  -H "Content-Type: application/json" http://127.0.0.1:4929/containers/nginx/exec \
  -d '{
    "AttachStdin": true,
    "AttachStdout": true,
    "AttachStderr": true,
    "Cmd": ["sh"],
    "DetachKeys": "ctrl-p,ctrl-q",
    "Privileged": true,
    "Tty": true,
    "User": "123:456"
  }'

{"Id":"3e1f26028f6bd72372c5cd1a3d2ce33a0531dc73ba70e69e94abde4a9564233"}
```

上面的参数与 Docker 的 exec 命令一致。这里只是创建了一个 exec，实际上还需要启动 exec。

7. 启动 exec

创建的 exec 默认不会启动，需要手动启动，格式如下：

```
POST /exec/(id)/start
```

示例 (curl) 如下：

```
$ curl -X POST \
  -H "Content-Type: application/json" \
```

```
http://127.0.0.1:4929/exec/3elf26028f6bd72372c5cda1a3d2ce33a0531dc73ba7
0e69e94abde4a9564233/start '{ "Detach": false, "Tty": false }'
```

输出示例：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.docker.raw-stream
{{ STREAM }}
```

8. 显示 Volume

显示容器的数据卷 (Volume)，格式如下：

```
GET /volumes
```

示例 (curl) 如下：

```
$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:4929/
volumes
```

指定数据卷：

```
{
  "Volumes": [
    {
      "Name": "tardis",
      "Driver": "local",
      "Mountpoint": "/var/lib/docker/volumes/tardis",
      "Labels": null,
      "Scope": "local"
    }
  ],
  "Warnings": []
}
```

本节的 API 介绍到此为止，后面的 API 实际上还有很多，例如创建数据卷、管理网络等的 API，这部分内容可以在官方文档中找到资料，地址为 https://docs.docker.com/engine/reference/api/docker_remote_api_v1.24/#misc。

16.3.2 Trusted Registry API 介绍

运行一个临时的 Registry 来学习 API。

```
$ docker run -d -p 5050:5000 --name reg_test registry:2.5.0
81ea280d60d5fb5b98b238919039772ea289ebcabf86431af2338f37677da629
```

推送一个镜像到 Registry 中。

```
$ docker tag nginx:alpine localhost:5050/library/nginx:alpine
$ dpush localhost:5050/library/nginx:alpine
The push refers to a repository [localhost:5050/library/nginx]
1bbf5c4f940b: Pushed
b36519590516: Pushed
da85d9b33770: Pushed
4fe15f8d0ae6: Pushed
alpine: digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c
6b30df7f23292fe size: 1154
```

测试连接。

```
$ curl -X GET http://localhost:5050/v2/
{}%
```

如果是需要认证的仓库，会返回：

```
$ curl -X GET -H "Content-Type: application/json" https://reg.example.com/
v2/
{"errors":[{"code":"UNAUTHORIZED","message":"authentication
required","detail":null}]}
```

现在可以使用 API 获取本地仓库的镜像信息了，例如获取指定镜像的全部标签：

```
$ curl -X GET http://localhost:5050/v2/library/nginx/tags/list
{"name":"library/nginx","tags":["alpine"]}
```

又如使用 Postman 获取镜像的 Manifest 信息，如图 16.11 所示。

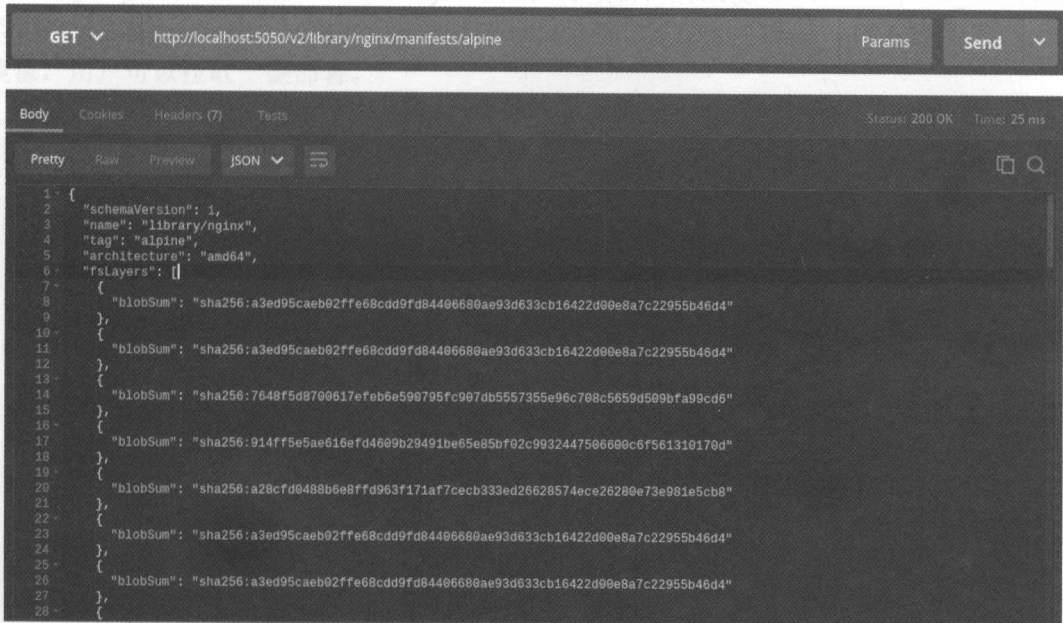


图 16.11 Postman 获取镜像的 Manifest

获取当前仓库的镜像目录如下：

```
$ curl -X GET http://localhost:5050/v2/_catalog
{"repositories":["library/nginx"]}
```

更详细的 API 资料可访问官方文档 <https://docs.docker.com/registry/spec/api/#/detail>。

16.4 本章小结

本章主要介绍了常见的 Docker API 的使用，以及 Docker API 的结构，目的在于帮助读者理解 Docker API 的集成与应用。在 Docker 生态系统中有很多工具都是使用 Docker API 与 Docker daemon 打交道的。

目前 Docker API 的更新依旧十分频繁，以后在参数和格式上可能有比较大的变动，本书在每节后面提供的链接是当前 Docker API 的最新版本，读者可以根据链接扩展相应的知识。

最后就是 Docker 官方提供的各种语言实现的 Docker API 工具，地址为 https://docs.docker.com/engine/reference/api/remote_api_client_libraries/。其中的项目可以作为不错的例子来学习 Docker API 的集成。

第 17 章 私有仓库

到目前为止，我们已经学习了 17 章的内容，在学习过程中想必读者也动手构建了许多自己的镜像，这些镜像保存在本地管理起来很不方便，推送到 Docker Hub 是不错的做法，但是 Docker Hub 免费版本只提供一个隐私镜像，其他镜像都是公开的，有时候我们不方便公开自己的镜像，又不想托管到第三方社区，只是希望与当前局域网的用户之间分享。

这种情况下就需要部署一个自己的私有仓库了，在前面第 7 章中介绍过如何在本地搭建一个 Registry，但那个 Registry 是远远不符合安全性要求的，而且只适合个人使用。在本章中，将实战搭建一个企业级的 Docker Registry。

17.1 Docker Registry 介绍

Registry(v1)在前期是一个基于 Python 的开源项目，后来使用 Go 重写了 Registry(v2)，代码托管在 <https://github.com/docker/distribution> 上面，官方也在 Docker Hub 提供了 Docker 镜像，用户可以拉取一键部署。

17.1.1 部署 Docker Registry

在第 7 章中，曾使用 `docker run` 命令直接运行了一个 Registry：

```
$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

这个容器没有指定配置文件，也没指定存储目录，容器删除后托管的镜像也会被删除。要指定配置文件路径很简单，通过如下命令即可。

```
$ docker run -d -p 5000:5000 \  
  -v [your/path/registry-conf]/:/registry-conf \  
  -e DOCKER_REGISTRY_CONFIG=/registry-conf/config.yml \  
  --restart=always --name registry registry:2
```

- `-v`：挂载配置文件目录。
- `-e`：指定配置目录路径。

还可以通过指定镜像存储位置并挂载，确保镜像资源不会丢失。

```
-v [your/path/registry]:/var/lib/registry
```

17.1.2 私有仓库的 push 与 pull

在 17.1.1 节搭建了一个本地的镜像仓库之后，可以向其推送镜像了，推送之前要先打标签，例如，推送 `nginx:alpie` 镜像到本地仓库。


```
$ docker tag nginx:alpine localhost:5050/library/nginx:alpine
$ docker push localhost:5050/library/nginx:alpine
```

这样一个本地镜像就推送到了本地镜像仓库中，现在可以使用 `pull` 命令来拉取这个镜像。

因为本地已经存在 `localhost:5050/library/nginx:alpine` 镜像，需先删除本地的镜像再拉取。

```
$ docker rmi localhost:5050/library/nginx:alpine
Untagged: localhost:5050/library/nginx:alpine
Untagged: localhost:5050/library/nginx@sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f23292fe
```

使用 `pull` 命令来拉取本地仓库的镜像。

```
$ docker pull localhost:5050/library/nginx:alpine
alpine: Pulling from library/nginx
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f23292fe
Status: Downloaded newer image for localhost:5050/library/nginx:alpine
```

17.1.3 配置 Registry

Registry 的配置文件也是一个 YAML 文件，在配置文件中定义的配置与在启动 Registry 时设置环境变量效果是一致的，例如配置文件中如下定义：

```
storage:
  filesystem:
    rootdirectory: /path/registry
```

上面定义了 Registry 的 `root` 目录，而使用 `docker run` 命令设置变量一样可以达到相同效果。

```
$ docker run -e REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY=/path/registry \
  -p 5000:5000 -d registry
```

上面变量指定的是镜像文件的存储位置，默认值是 `/var/lib/registry`，上面两种方式都可以重新定义该目录。

为了方便管理操作，一般使用挂载配置文件的方式配置 Registry。

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
  -v 'pwd'/config.yml:/etc/docker/registry/config.yml \
  registry:2
```

配置文件的写法将在后面展开来讲（下面是一整份完整的配置，为了方便讲解，将其拆分成了段落）。

```
# 首先是配置文件的版本，这个必须要有，而且放在第一行
version: 0.1
```

```
# 设置 Registry 的日志
```

```
# level 可以设置日志级别，可选的值为 error, warn, info, debug，非必需项
```

```
# formatter 选择日志的输出格式，可选值为 text, json, logstash。默认为 text，非必需项
```

```
# fields 设置日志输出时带有设定的标识，用于在与其他服务混合输出日志时便于阅读，非必需项
log:
```

```
  level: debug
```

```
  formatter: text
```

```
fields:
  service: registry
environment: staging
```

设置日志的 hooks 行为, 例如日志中出现 error 事件时发送邮件通知管理员

```
hooks:
  - type: mail
    disabled: true
    levels:
      - panic
    options:
      smtp:
        addr: mail.example.com:25
        username: mailuser
        password: password
        insecure: true
        from: sender@example.com
        to:
          - errors@example.com
```

设置文件存储

```
storage:
```

设置保存到本地的指定目录

```
filesystem:
  rootdirectory: /var/lib/registry
  maxthreads: 100
```

保存到 Azure 的容器服务中

```
azure:
  accountname: accountname
  accountkey: base64encodedaccountkey
```

```
container: containername
```

保存到 Google 的 GCS 中

```
gcs:
  bucket: bucketname
  keyfile: /path/to/keyfile
  rootdirectory: /gcs/object/name/prefix
```

```
chunksize: 5242880
```

保存到亚马逊的 S3 中

```
s3:
  accesskey: awsaccesskey
  secretkey: awssecretkey
  region: us-west-1
  regionendpoint: http://myobjects.local
  bucket: bucketname
  encrypt: true
  keyid: mykeyid
  secure: true
  v4auth: true
  chunksize: 5242880
```

```
rootdirectory: /s3/object/name/prefix
```

使用 Swift 保存

```
swift:
  username: username
  password: password
  authurl: https://storage.myprovider.com/auth/v1.0 or https://storage.
myprovider.com/v2.0 or https://storage.myprovider.com/v3/auth
  tenant: tenantname
  tenantid: tenantid
  domain: domain name for Openstack Identity v3 API
  domainid: domain id for Openstack Identity v3 API
```

```

    insecureSkipVerify: true
    region: fr
    container: containername
rootdirectory: /swift/object/name/prefix
# 使用阿里云的对象存储
oss:
    accesskeyid: accesskeyid
    accesskeysecret: accesskeysecret
    region: OSS region name
    endpoint: optional endpoints
    internal: optional internal endpoint
    bucket: OSS bucket
    encrypt: optional data encryption setting
    secure: optional ssl setting
    chunksize: optional size valye
rootdirectory: optional root directory
# 上面的设置可根据自己需要增删

# 下面这个参数表示使用内存作为存储，没有参数可以设置，仅测试用，非测试务必删除
inmemory: # 生产环境记得删除这个定义

# 设置 Registry 镜像是否允许删除
delete:
enabled: false
# 设置是否允许重定向
redirect:
disable: false
# 设置高速缓存驱动（缓存 Layer 元数据），可选的有 redis 与 inmemory
cache:
blobdescriptor: redis

# 启用维护状态（这个项目下面的参数全部为必需项）
# 下面的单位可以使用 45m, 2h10m, 168h (1 week) 这些格式
maintenance:
    uploadpurging:
        enabled: true          # 设置是否允许上传
        age: 168h              # 超过这个时间的镜像会被删除
        interval: 24h          # 多长时间清空仓库
        dryrun: false          # 设置为 true 可以获知哪些目录会被删除
        readonly:              # 可以设置为只读模式
        enabled: false

# 设置 Registry 的访问认证
# 下面认证设置其中一种即可
auth:
    # silly 只是检查存在的授权在 HTTP 请求 header，没有考虑 header 的值（不推荐）
    silly:
        realm: silly-realm
    service: silly-service
    # Docker Hub 采用此种方式 (https://docs.docker.com/registry/spec/auth/token/#/requesting-a-token)
    token:
        realm: token-realm
        service: token-service
        issuer: registry-token-issuer
rootcertbundle: /root/certs/bundle
# httpasswd 认证允许使用 Apache httpasswd 配置基本认证文件，配置简单（只支持 bcrypt 格式密码）

```

```

htpasswd:
  realm: basic-realm
path: /path/to/htpasswd

# 中间件设置
middleware:
  registry:
    - name: ARegistryMiddleware
      options:
        foo: bar
  repository:
    - name: ARepositoryMiddleware
      options:
        foo: bar
  storage:
    - name: cloudfront
      options:
        baseurl: https://my.cloudfronted.domain.com/
        privatekey: /path/to/pem
        keypairid: cloudfrontkeypairid
        duration: 3000s
  storage:
    - name: redirect
      options:
        baseurl: https://example.com/

# 错误报告是可选的，配置错误报告工具和指标
# 目前只支持两个服务：New Relic 和 Bugsnag，配置可以同时包含两个服务
reporting:
  bugsnag:
    apikey: bugsnagapikey
    releasestage: bugsnagreleasestage
    endpoint: bugsnagendpoint
  newrelic:
    licensekey: newreliclicensekey
    name: newrelicname
verbose: true

# 配置 HTTP 细节
http:
  addr: localhost:5000
  prefix: /my/nested/registry/
  host: https://myregistryaddress.org:5000
  secret: asecretforlocaldevelopment

# 设置 Registry 地址
# 设置 Registry 主机名称
# 一个随机的数据。防止篡改存储在客户端的签名状态

# 对于生产环境应该生成一个随机的数据
# secret 这个配置参数可以省略，在这种情况下 Registry 将自动生成一个并发送
relativeurls: false
# 配置 TLS
tls:
  certificate: /path/to/x509/public
key: /path/to/x509/private
# 自签名或者企业证书
clientcas:
  - /path/to/ca.pem
  - /path/to/another/ca.pem
# 使用 letsencrypt 签发的证书
letsencrypt:
  cachefile: /path/to/cache-file

```

```
email: emailused@letsencrypt.com

# 设置 Debug 的地址
debug:
  addr: localhost:5001
  headers:
X-Content-Type-Options: [nosniff]
# 配置通知, 涉及后面的知识点, 暂不讲
notifications:
  endpoints:
    - name: alistener
      disabled: false
      url: https://my.listener.com/event
      headers: <http.Header>
      timeout: 500
      threshold: 5
      backoff: 1000

# 配置 Redis
redis:
  addr: localhost:6379
  password: asecret
  db: 0
  dialtimeout: 10ms
  readtimeout: 10ms
  writetimeout: 10ms
  pool:
    maxidle: 16
    maxactive: 64
  idletimeout: 300s

# 健康检查
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
  file:
    - file: /path/to/checked/file
      interval: 10s
  http:
    - uri: http://server.to.check/must/return/200
      headers:
        Authorization: [Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==]
      statuscode: 200
      timeout: 3s
      interval: 10s
      threshold: 3
  tcp:
    - addr: redis-server.domain.com:6379
      timeout: 3s
      interval: 10s
      threshold: 3

# 设置代理
# 这个功能是 17.1.4 节要讲的内容, 搭建 Docker Hub Mirror
proxy:
  remoteurl: https://registry-1.docker.io
  username: [username]
  password: [password]
```

```
compatibility:
  schema1:
    signingkeyfile: /etc/registry/key.json
```

上面粗略介绍了全部配置定义，实际上一份配置文件中没用到的可不用写出来，因此真正一份配置文件并没有那么长，例如：

```
version: 0.1
log:
  level: debug
storage:
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: localhost:5000
  secret: asecretforlocaldevelopment
  debug:
    addr: localhost:5001
```

上面就定义了存储路径和 HTTP 一些细节。这样一份配置文件完全可以让 Registry 运行起来。

目前我们启动的 Registry 只能够拉取上传的镜像，17.1.4 节将添加“镜像仓库的镜像”功能。

17.1.4 添加 Docker Hub Mirror 功能

前面我们搭建的私有 Registry 只是一个本地的仓库，只能拉取自己推送上去的镜像，为了让本地 Registry 发挥更大作用，可以在配置文件中设置 Docker Hub Mirror，也就是说可以通过私有 Registry 来拉取 Docker Hub 的镜像。例如私有 Registry 中没有 Ubuntu 镜像，但是依旧可以使用 `docker pull` 命令从私有 Registry 中拉取 Ubuntu 镜像，因为私有仓库没有的镜像会从 Docker Hub 拉取，这就是 Mirror 的特点，利用这个功能，可以搭建一个自己用的镜像加速器。先来看如何配置 Mirror。

```
proxy:
  remoteurl: https://registry-1.docker.io
  username: [username]
  password: [password]
```

上面的 `username` 和 `password` 的值不是必须的，但是如果要访问 Docker Hub 上的私有镜像则必须要设置用户账号和密码。

下面使用 Docker Compose 启动一个带有 Docker Hub Mirror 功能的私有 Registry。

`docker-compose.yml` 如下：

```
server:
  image: registry:2.1.1
  ports:
    - "5000:5000"
  volumes:
    - ./config-mirror.yml:/etc/docker/registry/config-mirror.yml
    - ./data:/var/lib/registry
  command:
    - /etc/docker/registry/config-mirror.yml
config-mirror.yml:
version: 0.1
```

```
log:
  fields:
    service: registry
storage:
  cache:
    layerinfo: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: :5000
proxy:
  remoteurl: https://registry-1.docker.io
```

现在启动 Registry，然后在 docker daemon 启动时加上对应的--registry-mirror 即可，或者像之前基础部分讲到的设置 Mirror 地址也可以。

```
$ docker --registry-mirror=https://<my-docker-mirror-host> daemon
```

现在使用 docker pull 操作，就可以通过私有 Registry 拉取 Docker Hub 的镜像了。

17.2 认证与前端

截至前面的私有 Registry，都是可以直接访问推送镜像的，没有认证与权限管理。本节将带领大家完善这个私有 Registry。

17.2.1 设置反代理

首先，使用 IP 来拉取镜像显得很不舒服，不容易记忆。通常我们都会加一层反代理设置 Registry 的地址为一个域名，例如 reg.examole.com 等形式。

Nginx 的反代理相信大家都很熟悉，这里不做过多介绍，如不知道反代理是什么，可以粗略理解为“Nginx 反代理可以把我们的 Registry 仓库地址与指定域名连接起来，访问域名实际上是访问 IP:port”，本节直接基于上面的 Registry 演示。

```
server {
    listen 80;
    server_name myregistrydomain.com;

    # disable any limits to avoid HTTP 413 for large image uploads
    client_max_body_size 0;

    # required to avoid HTTP 411: see Issue #1486 (https://github.com/
docker/docker/issues/1486)
    chunked_transfer_encoding on;

    location /v2/ {
        # Do not allow connections from docker 1.5 and earlier
        # docker pre-1.6.0 did not properly set the user agent on ping, catch
        "Go *" user agents
        if (\$http_user_agent ~ "^(docker\/1\.(3|4|5(?:!|\. [0-9]-dev))|Go ).*\$" ) {
            return 404;
        }

        # To add basic authentication to v2 use auth_basic setting.
```



```

auth_basic "Registry realm";
auth_basic_user_file /etc/nginx/conf.d/nginx.htpasswd;

## If $docker_distribution_api_version is empty, the header will not
be added.
## See the map directive above where this variable is defined.
add_header 'Docker-Distribution-API-Version' \${docker_distribution_
api_version} always;
# 设置 reg.example.com 为用户自己的域名即可
proxy_pass http://reg.example.com;
proxy_set_header Host \${http_host}; # required for
docker client's sake
proxy_set_header X-Real-IP \${remote_addr}; # pass on real
client's IP
proxy_set_header X-Forwarded-For \${proxy_add_x_forwarded_for};
proxy_set_header X-Forwarded-Proto \${scheme};
proxy_read_timeout 900;
}
}

```

注意先别着急使用上面的配置来启动，因为即使上面的配置启动了，得到的也是一个不安全的“警告”。这是因为 Registry 强制使用 SSL 来保证数据传输安全，启用了域名之后如果没有 SSL 配置，Registry 是无法正常运行的（可以指定参数来取消 SSL 限制，但是不推荐）。

为了快速使用 HTTPS，可以使用 Caddy 服务器新建一个 Caddyfile。

```

reg.example.com {
  proxy / <Regsitry IP>:5000 {
    proxy_header Host {host}
    proxy_header X-Real-IP {remote}
    proxy_header X-Forwarded-Proto {scheme}
  }
  log /var/log/caddy.log
  gzip
}

```

保存然后启动 Caddy。

```

$ docker run -v ~/caddy/Caddyfile:/etc/Caddyfile \
-v ~/.caddy:/root/.caddy \
-p 80:80 -p 443:443 \
--name=caddy -d abiosoft/caddy

```

现在就可以使用更舒服的方式拉取镜像了：

```
$ docker pull reg.example.com/library/ubuntu
```

17.2.2 为私有仓库添加认证服务

到这里已完成了 Docker Hub Mirror、绑定域名并且设置反代理两个功能，但是现在还没有访问认证。添加认证有两种办法，一是使用 Registry 自带的认证功能，在前面配置文件中粗略介绍过，另一种办法是使用 Nginx 配置认证。

第一种办法是使用原生认证，首先新建一个 htpasswd 文件。

```

$ mkdir auth
$ docker run --entrypoint htpasswd registry:2 -Bbn testuser testpassword
> auth/htpasswd

```

然后启动一个 Registry（之前的仓库请停止）。

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
-v 'pwd'/auth:/auth \
-e "REGISTRY_AUTH=htpasswd" \
-e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
-e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
-v 'pwd'/certs:/certs \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
registry:2
```

现在需要登录才能访问 Registry。

```
$ docker login myregistrydomain.com:5000
```

上面的启动命令使用 Docker Compose 来管理会更方便。

```
registry:
  restart: always
  image: registry:2
  ports:
    - 5000:5000
  environment:
    REGISTRY_HTTP_TLS_CERTIFICATE: /certs/domain.crt
    REGISTRY_HTTP_TLS_KEY: /certs/domain.key
    REGISTRY_AUTH: htpasswd
    REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
    REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm
  volumes:
    - /path/data:/var/lib/registry
    - /path/certs:/certs
    - /path/auth:/auth
```

使用 `docker-compose up -d` 命令进行启动。

第二种办法是使用 Nginx 的认证（或者 Caddy），首先该办法的前提是已经获得 SSL 证书。

```
$ mkdir -p auth
$ mkdir -p data
```

在 Nginx 配置文件的相应 server 中添加以下内容：

```
location /v2/ {
    # Do not allow connections from docker 1.5 and earlier
    # docker pre-1.6.0 did not properly set the user agent on ping, catch
    "Go *" user agents
    if ($http_user_agent ~ "(docker\/1\.(3|4|5(?:!\.[0-9]-dev))|Go) .*") {
        return 404;
    }

    # To add basic authentication to v2 use auth_basic setting.
    auth_basic "Registry realm";
    auth_basic_user_file /etc/nginx/conf.d/nginx.htpasswd;

    ## If $docker_distribution_api_version is empty, the header will not
    be added.
    ## See the map directive above where this variable is defined.
    add_header 'Docker-Distribution-API-Version' $docker_distribution_
    api_version always;

    proxy_pass http://docker-registry;
    proxy_set_header Host $http_host; # required for docker
```

```

client's sake
    proxy_set_header    X-Real-IP          \${remote_addr}; # pass on real
client's IP
    proxy_set_header    X-Forwarded-For    \${proxy_add_x_forwarded_for};
    proxy_set_header    X-Forwarded-Proto \${scheme};
    proxy_read_timeout  900;
}

```

重点是加粗的内容，它们共同组成了一个简单的访问验证。现在使用同样的办法生成 `htpasswd`。

```

$ docker run --rm --entrypoint htpasswd registry:2 -bn testuser testpassword
> auth/nginx.htpasswd

```

复制证书到 `auth` 目录。

```

$ cp domain.crt auth
$ cp domain.key auth

```

最后创建 `docker-compose.yml` 文件如下：

```

docker-compose.yml
nginx:
  image: "nginx:1.9"
  ports:
    - 5043:443
  links:
    - registry:registry
  volumes:
    - ./auth:/etc/nginx/conf.d
    - ./auth/nginx.conf:/etc/nginx/nginx.conf:ro

registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  volumes:
    - 'pwd'./data:/var/lib/registry

```

使用 `docker-compose up -d` 命令启动即可。

现在访问私有 Registry 需要登录：

```

$ docker login \
  -u=testuser \
  -p=testpassword \
  -e=root@example.ch myregistrydomain.com:5043
$ docker tag ubuntu myregistrydomain.com:5043/test
$ docker push myregistrydomain.com:5043/test
$ docker pull myregistrydomain.com:5043/test

```

17.2.3 为私有仓库添加可视化界面

到了这里，我们的 Registry 已经有了 Mirror、反代、认证等功能，还启用了 HTTPS 访问，现在还缺一个可视化的界面，用于管理镜像。

Github 上有不少这类项目，我们选择其中一个作为演示，Github 地址为 <https://github.com/mkuchin/docker-registry-web>。

这里使用前面已经搭建好的 Registry。

```
$ docker run -it -p 8080:8080 \
  --name registry-web \
  --link <Your Registry Container Name> \
  -e REGISTRY_URL=http://registry-srv:5000/v2 \
  -e REGISTRY_NAME=localhost:5000 \
  hyper/docker-registry-web
```

如果用户用的是原生认证方案，可使用下面方式启动：

```
$ docker run -it -p 8080:8080 --name registry-web --link registry-srv \
  -e REGISTRY_URL=https://registry-srv:5000/v2 \
  -e REGISTRY_TRUST_ANY_SSL=true \
  -e REGISTRY_BASIC_AUTH="YWRtaW46Y2hhbmdlbWU=" \
  -e REGISTRY_NAME=localhost:5000 hyper/docker-registry-web
```

完成上面的步骤之后，在浏览器 <http://localhost:8080> 中就可以看到 Registry 界面了。

17.3 企业级私有仓库 Harbor

前面的 Registry 配置、部署是不是很烦琐？其实我们可以选择更轻松的方式部署私有 Registry。而 Harbor 就是一个一键搞定企业级 Registry 仓库部署的应用。

Harbor 是一个由 VMware 公司开发的，用于存储和分发 Docker 镜像的企业级 Registry 开源服务器，通过添加一些企业必需的功能特性，例如安全、标识和管理等，扩展了开源的 Docker Distribution。

作为一个企业级私有 Registry 服务器，Harbor 提供了更好的性能和安全性，提升用户使用 Registry 构建和运行环境传输镜像的效率。

Harbor 支持安装在多个 Registry 节点的镜像资源复制，镜像全部保存在私有 Registry 中，确保数据和知识产权在公司内部网络中管控。另外，Harbor 也提供了高级的安全特性，如用户管理，访问控制和活动审计等。

17.3.1 Harbor 配置详解

首先把 Harbor 的源代码仓库复制下来。

```
$ git clone https://github.com/vmware/harbor ~/harbor
```

然后编辑配置文件。

```
$ vim ~/harbor/Deploy/harbor.cfg
```

配置文件模板如下：

```
## Configuration file of Harbor

#The IP address or hostname to access admin UI and registry service.
#DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by
external clients.
#####
# 下面输入你的仓库网址，比如“reg.example.com”
# 不要使用 localhost 或者 127.0.0.1 作为 hostname
# 否则别人无法访问这个仓库
#####
hostname = reg.example.com
```

```

#The protocol for accessing the UI and token/notification service, by default
it is http.
#It can be set to https if ssl is enabled on nginx.
ui_url_protocol = https

#Email account settings for sending out password resetting emails.
#####
# 这里的设置可以无视, 只有密码找回才会用到
#####
email_server = smtp.mydomain.com
email_server_port = 25
email_username = sample_admin@mydomain.com
email_password = abc
email_from = admin <sample_admin@mydomain.com>

##The password of Harbor admin, change this before any production use.
#####
# 下面输入管理员密码
#####
harbor_admin_password = password

##By default the auth mode is db_auth, i.e. the credentials are stored in
a local database.
#Set it to ldap_auth if you want to verify a user's credentials against an
LDAP server.
auth_mode = db_auth

#The url for an ldap endpoint.
#####
# 可以不填
#####
ldap_url = ldaps://ldap.example.com

#The basedn template to look up a user in LDAP and verify the user's password.
#####
# 假设域名是 example.com, 所以这里填 dc=example,dc=com
#####
ldap_basedn = uid=%s,ou=people,dc=example,dc=com

#The password for the root user of mysql db, change this before any production
use.
#####
# 下面输入数据库密码
#####
db_password = password
#####
# 是否开放注册
#####
#Turn on or off the self-registration feature
self_registration = on

#Determine whether the UI should use compressed js files.
#For production, set it to on. For development, set it to off.
use_compressed_js = on

#Maximum number of job workers in job service
max_job_workers = 3

```

```
#The expiration of token used by token service, default is 30 minutes
token_expiration = 30

#Determine whether the job service should verify the ssl cert when it connects
to a remote registry.
#Set this flag to off when the remote registry uses a self-signed or untrusted
certificate.
verify_remote_cert = on

#Determine whether or not to generate certificate for the registry's token.
#If the value is on, the prepare script creates new root cert and private
key
#for generating token to access the registry. If the value is off, a
key/certificate must
#be supplied for token generation.
customize_cert = on
#####
# 用到 SSL , 这里填个人信息, 注意时区
#####
#Information of your organization for certificate
crt_country = CN
crt_state = GuangZhou
crt_location = CN
crt_organization = example
crt_organizationalunit = example
crt_commonname = example.com
crt_email = i@example.com
#####
```

完成配置之后, 使用脚本梳理检查, 然后准备下一步。

```
$ cd Deploy
$ ./prepare
Generated configuration file: ./config/ui/env
Generated configuration file: ./config/ui/app.conf
Generated configuration file: ./config/registry/config.yml
Generated configuration file: ./config/db/env
Generated configuration file: ./config/jobservice/env
Clearing the configuration file: ./config/ui/private_key.pem
Clearing the configuration file: ./config/registry/root.crt
Generated configuration file: ./config/ui/private_key.pem
Generated configuration file: ./config/registry/root.crt
The configuration files are ready, please use docker-compose to start the
service.
```

现在已经完成全部的 Harbor 配置工作。到这里其实可以直接运行了, 但是不建议, 因为会收到一个 Registry 没有加密传输的错误。下面配置 SSL 证书。

17.3.2 配置 HTTPS

首先看一下经过前面的配置之后, config 目录下都有什么文件, 注意 SSL 证书位置。

```
.
├── db
│   └── env
├── jobservice
│   ├── app.conf
│   └── env
└── nginx
```

```

|   |--- cert
|   |--- nginx.conf
|   |--- nginx.conf.bak
|   |--- nginx.https.conf
|--- registry
|   |--- config.yml
|   |--- root.crt      ## CRT 证书
|--- ui
|   |--- app.conf
|   |--- env
|   |--- private_key.pem  ## PEM 证书

```

6 directories, 11 files

如果要使用域名绑定私有仓库，必须开启 SSL。除了上面官方的自动生成自签名证书外，还可以使用更加正规的 SSL 证书，因为自签名证书在其他人的浏览器里会收到警告，虽然不影响使用。

下面使用 **letsencrypt** 获取证书：

```

$ git clone https://github.com/letsencrypt/letsencrypt
$ cd letsencrypt
$ ./letsencrypt-auto certonly -d reg.example.com

```

如图 17.1 所示，选择第二个选项，自动生成证书。

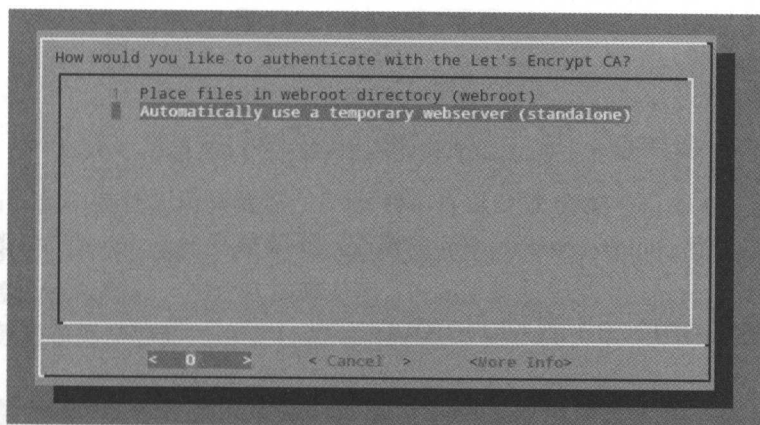


图 17.1 letsencrypt 获取 SSL 证书

如生成下面文字即为成功。

IMPORTANT NOTES:

```

- Congratulations! Your certificate and chain have been saved at
.....
.....
- If you like Let's Encrypt, please consider supporting our work by:
  Donating to ISRG / Let's Encrypt: https://letsencrypt.org/donate
  Donating to EFF: https://eff.org/donate-le

```

进入 Nginx 目录，配置 Nginx。

```
$ cd ~/harbor/Deploy/config/nginx
```

移动证书到 **cert/** 目录（这一步指的是自签名证书，使用 **letsencrypt** 申请的 SSL 证书存放目录在申请成功时有提示，移动到 **~/harbor/Deploy/config/nginx/cert/** 目录即可）。


```
$ cp ~/harbor/Deploy/config/registry/root.crt ~/harbor/Deploy/config/nginx/cert/
$ cp ~/harbor/Deploy/config/ui/private_key.pem ~/harbor/Deploy/config/nginx/cert/
```

备份原文件，使用 https 配置。

```
$ mv nginx.conf nginx.conf.bak && cp nginx.https.conf nginx.conf
```

然后修改 vim nginx.conf 文件，要改的地方很少：

```
.....

server {
    listen 443 ssl;
    # 下面改成你的域名
    server_name reg.example.com;

    # SSL
    # 使用 ./prepare 指令生成证书就这样填
    ssl_certificate /etc/nginx/cert/root.crt;
    ssl_certificate_key /etc/nginx/cert/private_key.pem;
    # SSL
    # 使用 letsencrypt 申请的参考这里
    # ssl_certificate /etc/nginx/cert/fullchain.pem;
    # ssl_certificate_key /etc/nginx/cert/privkey.pem;

    .....

server {
    listen 80;
    server_name reg.example.com;
    rewrite ^/(.*) https://$server_name$1 permanent;

    .....

```

这里使用了 80 端口、5000 端口以及 443 端口，需要确保这些端口没有占用，如果需要修改端口，要注意 docker-compose.yml 的修改，不要只在 nginx.conf 中修改。

17.3.3 使用 Compose 启动 Harbor

现在万事俱备，只需要使用 Compose 启动 Harbor 就可以部署 Registry 了。

```
$ cd ~/harbor/Deploy
$ docker-compose up
```

按 Enter 键之后，需要等待一段时间，如没有问题的话已经运行起来了。

```
$ docker-compose up
Starting deploy_log_1
Starting deploy_ui_1
Starting deploy_mysql_1
Starting deploy_registry_1
Starting deploy_jobservice_1
Starting deploy_proxy_1
Attaching to deploy_log_1, deploy_registry_1, deploy_ui_1, deploy_mysql_1,
deploy_jobservice_1, deploy_proxy_1
.....
mysql_1      | Version: '5.6.32' socket: '/var/run/mysqld/mysqld.sock'
port: 3306 MySQL Community Server (GPL)
ui_1         | 2016/09/01 23:59:31 [asm_amd64.s:1998] [I] http server Running
on :80
```

可以看到 Registry 服务器已经运行在 80 端口，表示服务已经运行起来，测试前需要绑定域名，域名生效后，就可以通过域名 pull 镜像了。

```
$ docker login reg.example.com
$ docker pull ubuntu
$ docker tag ubuntu reg.example.com/ubuntu
$ docker push reg.example.com/ubuntu
$ docker pull reg.example.com/ubuntu
```

如图 17.2 所示，现在可以使用 Harbor 界面管理镜像了。



图 17.2 Harbor 管理界面

17.4 私有仓库前端授权工具 Portus

Portus 是 SUSE 团队开发的，用于 Docker RegistryAPI（v2）的开源前端和授权工具，提供了更细粒度的权限控制、用户认证等功能。Portus 最低要求 Registry 版本是 2.1，它可以作为授权服务器和用户界面，用于 Docker Registry。

17.4.1 一键部署 Portus

Portus 提供了非常人性化的脚本来一键部署整个系统，前提是需要一台内存不低于 1 GB 的机器，否则 Portus 会因为内存不足而停止。

```
$ git clone https://github.com/SUSE/Portus.git ~/portus
$ cd portus && chmod a+x compose-setup.sh
$ ./compose-setup.sh
```

只需要一个脚本即可部署 Portus！

17.4.2 手动配置

和 Harbor 一样，依旧需要申请 SSL 证书，参考 Harbor 中的步骤即可，此处不再赘述。接下来创建一个 Registry 的配置文件，在其中指定刚才的证书和 token 方式的认证。

```
# config.yml
version: 0.1
loglevel: debug
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
delete:
  enabled: true
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
  tls:
    certificate: /certs/server-crt.pem
    key: /certs/server-key.pem

# 修改这里的 IP 地址为 Registry 容器宿主机的 IP 地址
auth:
  token:
    realm: https://<Host IP>/v2/token
    service: <Host IP>:5000
    issuer: <Host IP>
    rootcertbundle: /certs/server-crt.pem
notifications:
  endpoints:
    - name: portus
      url: https://<Host IP>/v2/webhooks/events
      timeout: 500ms
      threshold: 5
      backoff: 1s
```

然后就可以启动 Registry 容器了。

```
$ docker run -d \
  --name registry \
  -p 5000:5000 \
  --restart=always \
  -v /var/lib/registry:/var/lib/registry \
  -v /certs:/certs \
  -v /path/config.yml:/etc/docker/registry/config.yml \
  registry
```

17.4.3 启动 Portus

Docker Registry 配置完成后, 就该配置 Portus 了。Portus 需要数据库来存储信息, 虽然官方推荐 MariaDB, 但是 MySQL 也是没问题的。

启动数据库:

```
$ docker run -d \
  --name mariadb \
  --net=host \
  --restart=always \
  -e MYSQL_ROOT_PASSWORD=123456 \
  mariadb:10.1.10
```

数据库启动完成, 使用 `ecex` 进入容器并连接数据库。

```
$ docker exec -it mariadb mysql -uroot -p123456
```

为 Portus 创建用户和数据库。

```
create database portus;
GRANT ALL ON portus.* TO 'portus'@'%' IDENTIFIED BY 'portus';
exit
```

万事俱备，现在可以启动 Portus 了。

```
$ docker run -it -d \
  --name portus \
  --restart=always \
  -v /certs:/certs \
  --env DB_ADAPTER=mysql2 \
  --env DB_ENCODING=utf8 \
  --env DB_HOST=<Host IP> \
  --env DB_PORT=3306 \
  --env DB_USERNAME=portus \
  --env DB_PASSWORD=portus \
  --env DB_DATABASE=portus \
  --env RACK_ENV=production \
  --env RAILS_ENV=production \
  --env PUMA_SSL_KEY=/certs/server-key.pem \
  --env PUMA_SSL_CRT=/certs/server-crt.pem \
  --env PUMA_PORT=443 \
  --env PUMA_WORKERS=4 \
  --env MACHINE_FQDN=<Host IP> \
  --env SECRETS_SECRET_KEY_BASE=secret-goes-here \
  --env SECRETS_ENCRYPTION_PRIVATE_KEY_PATH=/certs/server-key.pem \
  --env SECRETS_PORTUS_PASSWORD=portuspw \
  h0tbird/portus:v2.0.2-1
```

启动完成后，在浏览器打开 <https://127.0.0.1/> 应该就能看到注册页面了。

17.5 本章小结

本章以 3 个实战例子为中心，从最基础的 Registry 仓库开始，逐步添加各项实用功能，讲解了私有 Registry 的配置与部署，最后使用 Harbor 和 Portus 作为实战项目，介绍了企业级 Registry 的部署过程。

通过本章的学习，相信读者已经能够搭建起自己的私有 Registry 服务。在生产环境中，还有更复杂的情况，例如分布式的 Registry 部署、多点容灾等的实现方案，这些内容都是本章没有介绍的。总之，学无止境，本章的内容只是一个稍微复杂的中级教程，更精彩的世界还需由读者自己来探索。

第 18 章 集群网络

前面已经介绍了基本的网络配置,相信读者已经对 Docker 网络相关的配置都有了一定的了解。本节将通过与 Swarm 结合,介绍跨主机多子网等复杂的网络配置方案。最后还会介绍一些著名的容器网络管理工具。

本章的主要内容有:

- 认识 Swarm。
- 配置跨主机容器网络。
- 认识容器网络管理工具。

18.1 Swarm 集 群

Swarm 在 Docker 1.12 版本之前属于一个独立的项目,在 Docker 1.12 版本发布之后,该项目合并到了 Docker 中,成为 Docker 的一个子命令。目前,Swarm 是 Docker 社区提供的唯一一个原生支持 Docker 集群管理的工具,可以把多个 Docker 主机组成的系统转换为单一的虚拟 Docker 主机,使得容器可以组成跨主机的子网网络。

18.1.1 认识 Swarm

在 Docker 1.12 版本之前,Docker 在集群管理上一直依靠第三方工具。以前的 Docker 服务自身只能在单台主机上进行操作,官方并没有真正意义上的集群管理方案。

直到 Docker 1.12 版本的发布,Docker 引擎在多主机、多容器的集群管理上才有了进一步的改进和完善,该版本的 Docker 内嵌了 swarm mode 集群管理模式。

为了方便演示跨主机网络,需要用到一个工具 Docker Machine,这个工具与 Docker Compose、Docker Swarm 并称 Docker 三剑客,下面来看看如何安装 Docker Machine。

```
$ sudo curl -L https://github.com/docker/machine/releases/download/v0.8.2/docker-machine-'uname -s'-'uname -m' > /usr/local/bin/docker-machine &&
chmod a+x /usr/local/bin/docker-machine
```

安装过程和 Docker Compose 非常类似。

现在 Docker 三剑客已经全部到齐,实战可以开始了。

在开始之前,需要了解一些基本概念,有关集群的 Docker 命令如下。

- docker swarm: 集群管理,子命令有 init、join、join-token、leave、update。
- docker node: 节点管理,子命令有 demote、inspect、ls、promote、rm、ps、update。
- docker service: 服务管理,子命令有 create、inspect、ps、ls、rm、scale、update。

- **docker stack/deploy**: 试验特性, 用于多应用部署 (本书不做介绍)。

首先使用 **Docker Machine** 创建一个虚拟机作为 **manager** 节点。

```
$ docker-machine create --driver virtualbox manager1
Running pre-create checks...
(manager1) Unable to get the latest Boot2Docker ISO release version: Get
https://api.github.com/repos/boot2docker/boot2docker/releases/latest:
dial tcp: lookup api.github.com on [::1]:53: server misbehaving
Creating machine...
(manager1) Unable to get the latest Boot2Docker ISO release version: Get
https://api.github.com/repos/boot2docker/boot2docker/releases/latest:
dial tcp: lookup api.github.com on [::1]:53: server misbehaving
(manager1) Copying /home/zuolan/.docker/machine/cache/boot2docker.iso to
/home/zuolan/.docker/machine/machines/manager1/boot2docker.iso...
(manager1) Creating VirtualBox VM...
(manager1) Creating SSH key...
(manager1) Starting the VM...
(manager1) Check network to re-create if needed...
(manager1) Found a new host-only adapter: "vboxnet0"
(manager1) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on
this virtual machine, run: docker-machine env manager1
```

然后查看虚拟机的环境变量等信息, 包括虚拟机的 IP 地址。

```
$ docker-machine env manager1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/home/zuolan/.docker/machine/machines/manager1"
export DOCKER_MACHINE_NAME="manager1"
# Run this command to configure your shell:
# eval $(docker-machine env manager1)
然后再创建一个节点作为 work 节点。
$ docker-machine create --driver virtualbox worker1
```

现在已经有了两个虚拟主机, 使用 **Machine** 的命令可以查看 (输出内容已调整):

```
$ docker-machine ls
NAME      ACTIVE DRIVER    STATE      URL                      DOCKER
manager1  -       virtualbox Running    tcp://192.168.99.100:2376 v1.12.3
worker1   -       virtualbox Running    tcp://192.168.99.101:2376 v1.12.3
```

但是目前这两台虚拟主机并没有什么联系, 为了把它们联系起来, 需要 **Swarm** 登场了。

因为我们使用的是 **Docker Machine** 创建的虚拟机, 因此可以使用 **docker-machine ssh** 命令来操作虚拟机, 在实际中, 并不需要像下面那样操作, 只需要执行 **docker swarm** 命令即可。

把 **manager1** 加入集群。

```
$ docker-machine ssh manager1 docker swarm init --listen-addr 192.168.99.100:2377 --advertise-addr 192.168.99.100
```

Swarm initialized: current node (23lkbq7uovqsg550qfzup59t6) is now a manager.

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token
SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwra kje
jql06klrfc585r \
192.168.99.100:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

用--listen-addr 指定监听的 IP 与端口, 实际的 Swarm 命令格式如下, 本例使用 Docker Machine 来连接虚拟机。

```
$ docker swarm init --listen-addr <MANAGER-IP>:<PORT>
```


接下来再把 work1 加入集群中。

```
$ docker-machine ssh worker1 docker swarm join --token \
```

```
SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwra kje
jql06klrfc585r \
192.168.99.100:2377
```

This node joined a swarm as a worker.

上面 join 命令中可以添加--listen-addr \$WORKER1_IP:2377 作为监听准备, 因为有时候可能会遇到把一个 work 节点提升为 manger 节点的可能, 当然本例中没有这个打算就不添加该参数了。

 **注意:** 如果在新建集群时遇到双网卡情况, 可以指定使用哪个 IP, 例如上面的例子。

```
$ docker-machine ssh manager1 docker swarm init --listen-addr $MANAGER1_
IP:2377
Error response from daemon: could not choose an IP address to advertise since
this system has multiple addresses on different interfaces (10.0.2.15 on
eth0 and 192.168.99.100 on eth1) - specify one with --advertise-addr
exit status 1
```

发生错误的原因是因为有两个 IP 地址, 而 Swarm 不知道用户想使用哪个, 因此要指定 IP。

```
$ docker-machine ssh manager1 docker swarm init --advertise-addr 192.
168.99.100 --listen-addr 192.168.99.100:2377
Swarm initialized: current node (ahvwxicunj d0z8g0eeosjztjx) is now a
manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token
SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-c036gwra kje
jql06klrfc585r \
192.168.99.100:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

集群初始化成功。

现在已经新建了一个有两个节点的“集群”，然后进入其中一个管理节点使用 `docker node` 命令来查看节点信息。

```
$ docker-machine ssh manager1 docker node ls
ID                HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
23lkbq7u... *   manager1  Ready   Active        Leader
dqb3fim8...      worker1  Ready   Active
```

现在每个节点都归属于 `Swarm`，并都处在了待机状态。`Manager1` 是领导者，`work1` 是工人。

现在继续新建虚拟机 `manger2`、`worker2`、`worker3`，现在已经有五个虚拟机了，使用 `docker-machine ls` 来查看虚拟机（输出已经调整）。

NAME	ACTIVE	DRIVER	STATE	URL	DOCKER
manager1	-	virtualbox	Running	tcp://192.168.99.100:2376	v1.12.3
manager2	-	virtualbox	Running	tcp://192.168.99.105:2376	v1.12.3
worker1	-	virtualbox	Running	tcp://192.168.99.102:2376	v1.12.3
worker2	-	virtualbox	Running	tcp://192.168.99.103:2376	v1.12.3
worker3	-	virtualbox	Running	tcp://192.168.99.104:2376	v1.12.3

然后把剩余的虚拟机也加到集群中。

添加 `worker2` 到集群中。

```
$ docker-machine ssh worker2 docker swarm join \
  --token SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-
c036gwrakjejql06klrfc585r \
  192.168.99.100:2377
This node joined a swarm as a worker.
```

添加 `worker3` 到集群中。

```
$ docker-machine ssh worker3 docker swarm join \
  --token SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-
c036gwrakjejql06klrfc585r \
  192.168.99.100:2377
This node joined a swarm as a worker.
```

添加 `manager2` 到集群中。

(1) 先从 `manager1` 中获取 `manager` 的 `token`。

```
$ docker-machine ssh manager1 docker swarm join-token manager
To add a manager to this swarm, run the following command:
```

```
docker swarm join \
  --token SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-
8tn855hkjdb6usrblo9iu700o \
  192.168.99.100:2377
```

(2) 然后添加 `manager2` 到集群中。

```
$ docker-machine ssh manager2 docker swarm join \
  --token
SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-8tn855hkjdb
6usrblo9iu700o \
  192.168.99.100:2377
This node joined a swarm as a manager.
```

(3) 然后再来查看集群信息。

```
$ docker-machine ssh manager2 docker node ls
ID                HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
16w80jnqy2k30yez4wbbaz1l8  worker1  Ready   Active
```

```

2gkwhzakejj72n5xoxruet71z    worker2    Ready    Active
35kutfynlratc55fn7j3fs4x      worker3    Ready    Active
a9r2lg5iqlu6h3lmyprfwl8ln *   manager2    Ready    Active    Reachable
dpo7snxbz2a0dxvx6mf19p35z     manager1    Ready    Active    Leader

```

18.1.2 建立跨主机网络

为了演示更清晰，下面把宿主机也加入到集群之中，这样使用 Docker 命令操作会清晰很多。

直接在本地执行加入集群命令：

```

$ docker swarm join \
  --token SWMTKN-1-3z5rzoey0u6onkvvm58f7vgkser5d7z8sfshlu7s4oz2gztlvj-
  8tn855hkjdb6usrblo9iu700o \
  192.168.99.100:2377
This node joined a swarm as a manager.

```

现在我们有 3 台 manager，3 台 worker。其中有 1 台是宿主机，另外 5 台是虚拟机。

```

$ docker node ls
ID                HOSTNAME        STATUS    AVAILABILITY  MANAGER STATUS
6zzrpk1t...      worker3         Ready     Active
7qbr0xd... *     user-pc         Ready     Active         Reachable
9v93sav...       manager2        Ready     Active         Reachable
alner3z...       worker1         Ready     Active
a5w7hj...        worker2         Ready     Active
d4h7vue...       manager1        Ready     Active         Leader

```

查看网络状态如下：

```

$ docker network ls
NETWORK ID        NAME        DRIVER    SCOPE
764ff31881e5     bridge     bridge    local
fbd9a977aa03     host       host      local
6p6xlousvsy2     ingress    overlay    swarm
e81af24d643d     none      null      local

```

可以看到在 swarm 上默认已有一个名为 ingress 的 overlay 网络，默认在 swarm 里使用，本例中会创建一个新的 overlay 网络。

```

$ docker network create --driver overlay swarm_test
4dm8cy9y5delvs5vd0ghdd89s
$ docker network ls
NETWORK ID        NAME        DRIVER    SCOPE
764ff31881e5     bridge     bridge    local
fbd9a977aa03     host       host      local
6p6xlousvsy2     ingress    overlay    swarm
e81af24d643d     none      null      local
4dm8cy9y5del     swarm_test  overlay    swarm

```

这样一个跨主机网络就搭建好了，但是现在这个网络只是处于待机状态，18.1.3 节我们会在这个网络上部署应用。

18.1.3 在跨主机网络上部署应用

首先前面创建的节点都是没有镜像的，因此要逐一 pull 镜像到节点中，这里使用前面搭建的私有仓库。

```

$ docker-machine ssh manager1 docker pull reg.example.com/library/nginx:
alpine
alpine: Pulling from library/nginx
e110a4a17941: Pulling fs layer
.....
7648f5d87006: Pull complete
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f
23292fe
Status: Downloaded newer image for reg.example.com/library/nginx:alpine
$ docker-machine ssh manager2 docker pull reg.example.com/library/nginx:
alpine
alpine: Pulling from library/nginx
e110a4a17941: Pulling fs layer
.....
7648f5d87006: Pull complete
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f
23292fe
Status: Downloaded newer image for reg.example.com/library/nginx:alpine
$ docker-machine ssh worker1 docker pull reg.example.com/library/nginx:
alpine
alpine: Pulling from library/nginx
e110a4a17941: Pulling fs layer
.....
7648f5d87006: Pull complete
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f
23292fe
Status: Downloaded newer image for reg.example.com/library/nginx:alpine
$ docker-machine ssh worker2 docker pull reg.example.com/library/nginx:
alpine
alpine: Pulling from library/nginx
e110a4a17941: Pulling fs layer
.....
7648f5d87006: Pull complete
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f
23292fe
Status: Downloaded newer image for reg.example.com/library/nginx:alpine
$ docker-machine ssh worker3 docker pull reg.example.com/library/nginx:
alpine
alpine: Pulling from library/nginx
e110a4a17941: Pulling fs layer
.....
7648f5d87006: Pull complete
Digest: sha256:65063cb82bf508fd5a731318e795b2abbfb0c2222f02ff5c6b30df7f
23292fe
Status: Downloaded newer image for reg.example.com/library/nginx:alpine

```

上面使用 `docker pull` 分别在 5 个虚拟机节点拉取 `nginx:alpine` 镜像。接下来要在 5 个节点部署一组 Nginx 服务。

部署的服务使用 `swarm_test` 跨主机网络。

```

$ docker service create \
  --replicas 2 \
  --name helloworld \
  --network=swarm_test nginx:alpine
5gz0h2s5agh2d2libvzq6bhgs

```

查看服务状态如下：

```

$ docker service ls
ID            NAME            REPLICAS  IMAGE            COMMAND
5gz0h2s5agh2 helloworld      0/2       nginx:alpine

```

查看 helloworld 服务详情（为了方便阅读，已调整输出内容）如下：

```
$ docker service ps helloworld
ID            NAME            IMAGE            NODE            DESIRED STATE  CURRENT STATE      seconds ago
ay081uome3    helloworld.1    nginx:alpine     manager1        Running         Preparing 2
seconds ago
16cvore0c96    helloworld.2    nginx:alpine     worker2         Running         Preparing 2
seconds ago
```

可以看到两个实例分别运行在两个节点上。

进入两个节点，查看服务状态（为了方便阅读，已调整输出内容）如下：

```
$ docker-machine ssh manager1 docker ps -a
CONTAINER ID   IMAGE            COMMAND          CREATED        STATUS        PORTS          NAMES
119f787622c2   nginx:alpine     "nginx -g ..." 4 minutes ago  Up 4 minutes  80/tcp, 443/tcp  hello ...

$ docker-machine ssh worker2 docker ps -a
CONTAINER ID   IMAGE            COMMAND          CREATED        STATUS        PORTS          NAMES
5db707401a06   nginx:alpine     "nginx -g ..." 4 minutes ago  Up 4 minutes  80/tcp, 443/tcp  hello ...
```

上面输出做了调整，实际的 NAMES 值为：

- helloworld.1.ay081uome3eejeg4mspa8pdlx;
- helloworld.2.16cvore0c96rbylvp0sny3mvt。

记住上面这两个实例的名称。现在来看这两个跨主机的容器是否能互通：

首先使用 Machine 进入 manager1 节点，然后使用 docker exec -i 命令进入 helloworld.1 容器中 ping 运行在 worker2 节点的 helloworld.2 容器。

```
$ docker-machine ssh manager1 \
  docker exec -i helloworld.1.ay081uome3eejeg4mspa8pdlx \
  ping helloworld.2.16cvore0c96rbylvp0sny3mvt
PING helloworld.2.16cvore0c96rbylvp0sny3mvt (10.0.0.4): 56 data bytes
64 bytes from 10.0.0.4: seq=0 ttl=64 time=0.591 ms
64 bytes from 10.0.0.4: seq=1 ttl=64 time=0.594 ms
64 bytes from 10.0.0.4: seq=2 ttl=64 time=0.624 ms
64 bytes from 10.0.0.4: seq=3 ttl=64 time=0.612 ms
^C
```

然后使用 Machine 进入 worker2 节点，使用 docker exec -i 命令进入 helloworld.2 容器中 ping 运行在 manager1 节点的 helloworld.1 容器。

```
$ docker-machine ssh worker2 \
  docker exec -i helloworld.2.16cvore0c96rbylvp0sny3mvt \
  ping helloworld.1.ay081uome3eejeg4mspa8pdlx
PING helloworld.1.ay081uome3eejeg4mspa8pdlx (10.0.0.3): 56 data bytes
64 bytes from 10.0.0.3: seq=0 ttl=64 time=0.466 ms
64 bytes from 10.0.0.3: seq=1 ttl=64 time=0.465 ms
64 bytes from 10.0.0.3: seq=2 ttl=64 time=0.548 ms
64 bytes from 10.0.0.3: seq=3 ttl=64 time=0.689 ms
^C
```

可以看到这两个跨主机的服务集群里面各个容器是可以互相连接的。

为了体现 Swarm 集群的优势，可以使用虚拟机的 ping 命令来测试对方虚拟机内的容器。

```
$ docker-machine ssh worker2 \
  ping helloworld.1.ay081uome3eejeg4mspa8pdlx
PING helloworld.1.ay081uome3eejeg4mspa8pdlx (221.179.46.190): 56 data bytes
64 bytes from 221.179.46.190: seq=0 ttl=63 time=48.651 ms
64 bytes from 221.179.46.190: seq=1 ttl=63 time=63.239 ms
64 bytes from 221.179.46.190: seq=2 ttl=63 time=47.686 ms
```

```

64 bytes from 221.179.46.190: seq=3 ttl=63 time=61.232 ms
^C
$ docker-machine ssh manager1 \
  ping helloworld.2.16cvore0c96rbylvp0sny3mvt
PING helloworld.2.16cvore0c96rbylvp0sny3mvt (221.179.46.194): 56 data bytes
64 bytes from 221.179.46.194: seq=0 ttl=63 time=30.150 ms
64 bytes from 221.179.46.194: seq=1 ttl=63 time=54.455 ms
64 bytes from 221.179.46.194: seq=2 ttl=63 time=73.862 ms
64 bytes from 221.179.46.194: seq=3 ttl=63 time=53.171 ms
^C

```

上面使用了虚拟机内部的 ping 去测试容器的延迟，可以看到延迟明显比集群内部的 ping 值要高。

为什么会造成这种现象呢？在前面网络基础中讲过 overlay 网络模型，从图 18.1 中可以看到，容器之间互 ping 属于一个子网的操作，因此 ping 值不高；而使用虚拟机内部的 ping 命令去测试时相当于另一个网络的访问，延迟就高了很多，尽管它们的物理地址都是同一个地点。

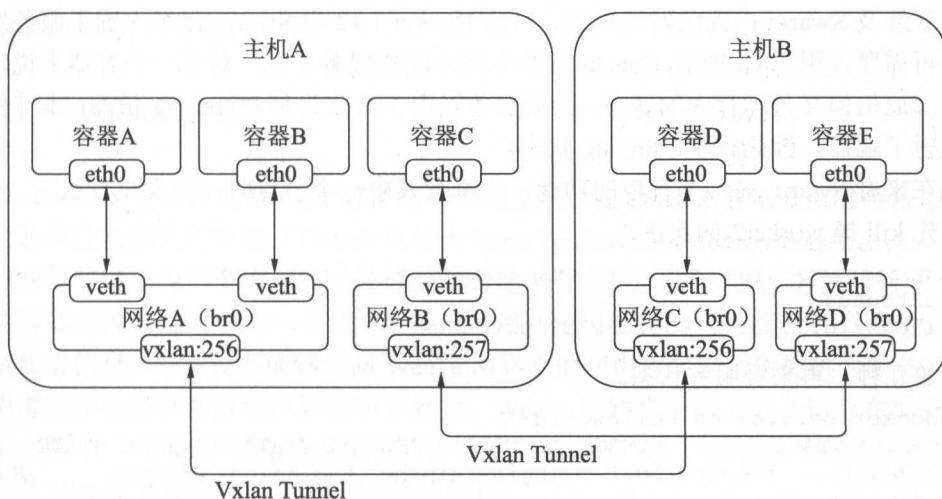


图 18.1 overlay 网络的结构示意

18.1.4 Swarm 集群负载

我们已经学会了 Swarm 集群的部署方法，现在来搭建一个可访问的 Nginx 集群吧。体验最新版的 Swarm 所提供的自动服务发现与集群负载功能。

首先删掉前面启动的 helloworld 服务。

```

$ docker service rm helloworld
helloworld

```

然后再新建一个服务，提供端口映射参数，使得外界可以访问这些 Nginx 服务。

```

$ docker service create \
  --replicas 2 \
  --name helloworld \
  -p 7080:80 \
  --network=swarm_test nginx:alpine
9gfziifbii7a6zdt56kocyun

```

查看服务运行状态:

```
$ docker service ls
ID                NAME                REPLICAS    IMAGE                COMMAND
9gfsziifbii7a    helloworld         2/2         nginx:alpine
```

不知读者是否发现, 虽然我们使用 `--replicas` 参数的值都是一样的, 但是前面获取服务状态时, `REPLICAS` 返回的是 `0/2`, 现在的 `REPLICAS` 返回的是 `2/2`。

同样使用 `docker service ps` 查看服务详细状态时(下面输出已经手动调整为更易读的格式), 可以看到实例的 `CURRENT STATE` 中是 `Running` 状态的, 而前面的 `CURRENT STATE` 中全部是处于 `Preparing` 状态。

```
$ docker service ps helloworld
ID                NAME                IMAGE                NODE                DESIRED STATE    CURRENT STATE    ERROR
9ikr3agyi...     helloworld.1       nginx:alpine        user-pc            Running          Running
13 seconds ago
7acmhj0u...     helloworld.2       nginx:alpine        worker2            Running          Running 6
seconds ago
```

这就涉及 `Swarm` 内置的发现机制, 目前 `Docker 1.12` 中 `Swarm` 已经内置了服务发现工具, 不再需要使用 `Etcd` 或者 `Consul` 这些工具来配置服务发现。对于一个容器来说, 如果没有外部通信但又是运行中的状态, 会被服务发现工具认为是 `Preparing` 状态, 本节例子中因为映射了端口, 因此有了 `Running` 状态。

现在来看 `Swarm` 另一个有趣的功能, 当杀死其中一个节点时, 会发生什么。

首先 `kill` 掉 `worker2` 的实例。

```
$ docker machine ssh worker2 docker kill helloworld.2.7acmhj0udzusvld
7lu2tbuhu4
helloworld.2.7acmhj0udzusvld7lu2tbuhu4
```

稍等几秒, 再来看服务状态。

```
$ docker service ps helloworld
ID                NAME                IMAGE                NODE                DESIRED STATE    CURRENT STATE    ERROR
9ikr3agyi...     helloworld.1       nginx:alpine        zuolan-pc          Running          Running
19 minutes ago
8f866igpl...     helloworld.2       nginx:alpine        manager1           Running          Running 4
seconds ago
7acmhj0u...     \_ helloworld.2    nginx:alpine        worker2            Shutdown        Failed 11
seconds ago ...exit...
```

```
$ docker service ls
ID                NAME                REPLICAS    IMAGE                COMMAND
9gfsziifbii7a    helloworld         2/2         nginx:alpine
```

可以看到即使 `kill` 掉其中一个实例, `Swarm` 也会迅速把停止的容器撤下来, 同时在节点中启动一个新的实例顶上来。这样服务依旧还是两个实例在运行。

此时如果想添加更多实例, 可以使用 `scale` 命令:

```
$ docker service scale helloworld=3
helloworld scaled to 3
```

查看服务详情, 可以看到有 3 个实例启动了:

```
$ docker service ps helloworld
ID                NAME                IMAGE                NODE                DESIRED STATE    CURRENT STATE    ERROR
9ikr3agyi...     helloworld.1       nginx:alpine        user-pc            Running          Running 30
minutes ago
8f866igpl...     helloworld.2       nginx:alpine        manager1           Running          Running
Running 11 minutes ago
7acmhj0u...     \_ helloworld.2    nginx:alpine        worker2
```



```
Shutdown      Failed 11 minutes ago  exit137
lvexrljm...   helloworld.3  nginx:alpine  worker2  Running  Running
4 seconds ago
```

现在如果想减少实例数量，一样可以使用 `scale` 命令：

```
$ docker service scale helloworld=2
helloworld scaled to 2
```

18.2 第三方网络管理工具

在 Docker overlay 网络模型出来之前，Docker 的跨主机管理都是依靠第三方工具实现的，因此社区出现了一些比较成熟的网络管理方案。

Docker 发布了 overlay 网络模式之后，这些网络工具逐渐消失在人们视野中，但是依旧有一些工具还在发挥余热，或者转型为范围更广的容器网络管理工具。本节将针对三款比较出名的 SDN 工具进行介绍。

18.2.1 Weave 介绍

Weave 是一个开源的 Docker 容器网络管理工具，在 Docker 网络未成熟阶段展现了其良好的易用性与强大的功能（Github 地址为 <https://github.com/weaveworks/weave>）。

Weave 能够创建一个虚拟网络来连接部署在多台主机上的 Docker 容器。通过 Weave，所有的容器就像被接入了同一个网络交换机，那些使用网络的应用程序不必去配置端口映射和链接等信息。外部设备能够访问 Weave 网络上的应用程序容器所提供的服务，同时已有的内部系统也能够暴露到应用程序容器上。Weave 能够穿透防火墙并运行在部分连接的网络上。另外，Weave 的通信支持加密，所以用户可以从一个不受信任的网络连接到主机。

安装 Weave 非常简单，使用 `curl` 下载即可。

```
$ sudo curl -L git.io/weave -o /usr/local/bin/weave
$ sudo chmod a+x /usr/local/bin/weave
```

上面的安装要在全部节点中执行，然后在其中一个节点启动 Weave。

```
$ weave launch
Unable to find image 'weaveworks/weaveexec:1.7.2' locally
1.7.2: Pulling from weaveworks/weaveexec
c0cb142e4345: Already exists
.....
b8e5c537a426: Pull complete
Digest: sha256:a91a8d45964d4a40c93192dc08d9f85b08013e6ddb8148c553207a9d55bc8e9b
Status: Downloaded newer image for weaveworks/weaveexec:1.7.2
Unable to find image 'weaveworks/weavedb:latest' locally
latest: Pulling from weaveworks/weavedb
66670338eee5: Pulling fs layer
.....
66670338eee5: Pull complete
Digest: sha256:6b1d2ca288a3a1e5551d99c419baf84c7b65948d5edc06a6cf8c9630f6d3cdf
Status: Downloaded newer image for weaveworks/weavedb:latest
Unable to find image 'weaveworks/weave:1.7.2' locally
```



```

1.7.2: Pulling from weaveworks/weave
a3ed95caeb02: Pulling fs layer
.....
5c2e78a9d0e0: Pull complete
Digest: sha256:da0e9c612e67ab6ab6aef263c279ed00a732f9473ebee3e057e11d92e
adcf74b
Status: Downloaded newer image for weaveworks/weave:1.7.2
Unable to find image 'weaveworks/plugin:1.7.2' locally
1.7.2: Pulling from weaveworks/plugin
c0cb142e4345: Already exists
.....
a3ed95caeb02: Pull complete
Digest: sha256:73b0a17e258febfd50906f943d71699e427885af49b637bcfdd0ae06c
3f208aa
Status: Downloaded newer image for weaveworks/plugin:1.7.2

```

启动之后一共拉取了 4 个镜像，使用 `docker ps` 查看后可以看到创建了 5 个容器，分别是 `weaveplugin`、`weaveproxy`、`weave`、`weavevolumes-1.7.2` 和 `weavedb`，其中后面两个容器只是创建没有运行。

启动之后使用 `Weave` 启动一个容器。

```
$ weave run 10.0.1.2/24 -it nginx:alpine
e54t95dasb02
```

下面用两台主机来连接建立跨主机的子网。

上面已经在主机 A 启动了 `Weave`，接下来在主机 B 执行：

```
$ weave launch <主机 A IP>
$ weave run 10.0.1.2/24 -it nginx:alpine
w3fe4f3r43e32
```

在启动 `nginx:alpine` 容器时指定了与第一个容器相同子网内的不同 IP。

然后在主机 A 的容器中 `ping` 主机 B 的容器。

```
[yangdong@centos7-A ~]$ docker attach e54t95dasb02
root@e54t95dasb02:/# ping 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=1 ttl=64 time=5.43 ms
64 bytes from 10.0.1.3: icmp_seq=2 ttl=64 time=0.808 ms
64 bytes from 10.0.1.3: icmp_seq=3 ttl=64 time=1.29 ms
^C
--- 10.0.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.808/2.513/5.435/2.075 ms
```

在主机 B 的容器中 `ping` 主机 A 的容器。

```
[yangdong@centos7-B ~]$ docker attach w3fe4f3r43e32
root@w3fe4f3r43e32:/# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=13.8 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=0.721 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=0.761 ms
^C
--- 10.0.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.721/5.099/13.815/6.163 ms
```

可以看到，这两个容器已经在同一个子网中。`Weave` 的介绍就到这里，更多的用法可参考官方文档。

18.2.2 Flannel 介绍

Flannel 是由 CoreOS 团队针对 Kubernetes 设计的一个覆盖网络工具，其目的在于帮助每一个使用 Kuberentes 的 CoreOS 主机拥有一个完整的子网。Kubernetes 会为每一个 POD 分配一个独立的 IP 地址，这样便于同一个 POD 中的 Containers 彼此连接，而之前的 CoreOS 并不具备这种能力。为了解决这一问题，Flannel 通过在集群中创建一个覆盖网络为主机设定一个子网（Github 地址为 <https://github.com/coreos/flannel>）。

Flannel 的编译比较简单，从 github 上下载到本地后直接构建即可，但目前 Flannel 仅支持 Linux，在其他平台编译时会报错。

```
$ git clone https://github.com/coreos/flannel.git
$ cd flannel
$ ./build
```

为创建全局唯一的 IP 地址，Flannel 需要用 Etcd 来存储每台机器上的子网地址。因此在启动 Flannel 之前，需要安装并配置 Etcd（Swarm 不用配置这些是因为其内置了服务发现机制）。

```
$ git clone https://github.com/coreos/etcd.git
$ ./build
$ setsid ./etcd \
  --listen-client-urls http://0.0.0.0:4001 \
  --advertise-client-urls http://k82demo1:4001 2>&1 > etcd.log &
```

在上面安装并启动 Etcd 之后，就可以配置 Flannel 了。

使用下面命令配置一个默认的子网。

```
$ etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
```

更多 Flannel 的配置参见其官网。

配置完成后即可启动 Flannel 和 Docker。Flannel 会与 Etcd 通信确定当前机器的子网，并将子网信息写入 `/run/flannel/subnet.env`，为了让 Docker 使用 Flannel 配置的子网，需要重启 Docker 并配置 `--bip` 参数。

Flannel 更多时候是在 Kubernetes 网络中的应用，在后面的章节中会介绍到 Kubernetes。

18.2.3 Pipework 介绍

Pipework 是由 Docker 的一个工程师设计的解决方案，它让容器能够在“任意复杂的场景”下进行连接。Pipework 是 Docker 的一个网络功能增强插件，它使用了 `cgroups` 和 `namespacpace`（Github 地址为 <https://github.com/jpetazzo/pipework>）。

Pipework 的项目目前处于不活跃开发状态，毕竟 Swarm 已经整合，而 Pipework 本身也有 Docker 背景，因此如果使用 Pipework 管理跨主机网络的话不如使用 Swarm。

本节以一个跨主机的 LAMP 部署例子，说明 Pipework 的网络管理过程。

安装 Pipework:

```
$ wget https://github.com/jpetazzo/pipework/archive/master.zip
$ unzip pipework-master.zip
$ cp pipework-master/pipework /usr/local/bin/
```

```
$ chmod +x /usr/local/bin/pipework
```

启动两个容器，一个 Apache 和一个 MySQL 容器，并为它们分配子网 IP。

```
$ APACHE=$(docker run -d apache /usr/sbin/httpd -D FOREGROUND)
```

```
$ MYSQL=$(docker run -d mysql /usr/sbin/mysqld_safe)
```

```
$ pipework br1 $APACHE 192.168.1.1/24
```

```
$ pipework br1 $MYSQL 192.168.1.2/24
```

Pipework 一般配合 Open vSwitch 使用，因为目前 Docker 已经支持 overlay 网络模型，本章不再介绍 Open vSwitch 项目，如果读者有兴趣可以在 <http://www.openvswitch.org/> 中阅读更多资料。

18.3 第三方服务发现

18.3.1 Etcd 介绍

在 Docker 中，Swarm Manager 是集成了服务发现机制的，因此用户使用 Swarm 部署集群时不需要再安装配置 k-v 数据库。

在 Docker 没有集成 Swarm 前，一般使用 Swarm 配合其他服务发现工具，例如 Etcd 等实现集群部署管理。

安装配置 Etcd 非常简单，只需要启动一个容器即可。

```
$ docker run -d -v /etc/ssl/certs:/etc/ssl/certs -p 4001:4001 -p 2380:2380
-p 2379:2379 \
--name etcd ystyle/etcd \
-name etcd0 \
-advertise-client-urls http://${HOSTIP}:2379,http://${HOSTIP}:4001 \
-listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
-initial-advertise-peer-urls http://${HOSTIP}:2380 \
-listen-peer-urls http://0.0.0.0:2380 \
-initial-cluster-token etcd-cluster-1 \
-initial-cluster etcd0=http://${HOSTIP}:2380 \
-initial-cluster-state new
```

其他节点加入集群时只需要执行：

```
$ docker run -d swarm join --addr=${NODEIP}:2375 etcd://${HOSTIP}:2379/swarm
```

完成节点添加之后，在管理节点启动 Swarm Manager。

```
$ docker run -d -p 3376:3376 -t \
swarm manage \
-H 0.0.0.0:3376 \
etcd://192.168.99.100:2379/swarm
```

如果没有异常，现在可以查看 Swarm 的节点列表。

```
$ docker run --rm swarm list etcd://192.168.99.100:2379/swarm
```

启动一个 Nginx 容器测试集群状态。

```
$ docker run --rm -p 8080:80 nginx:alpine
```

查看 Nginx 安装到哪台机器上。

```
$ docker ps -a
```

18.3.2 Consul 介绍

Consul 是一款服务注册发现的软件，自身是一个 key/value 的 store。在 Docker 1.12 发布之前，许多人选择用 Consul 和 Docker 结合起来提供一个高可扩展性的 Web 服务。

开始实验前要先修改 Docker 的主配置文件，使用 Consul 替换默认的 Docker 自身的 key/value store 中心。

```
ExecStart=/usr/bin/dockerd -H unix:///var/run/docker.sock --cluster-store=
consul://${HOSTIP}:8500
```

然后启动 Consul 容器。

```
$ docker run -d --restart=always \
  -h node \
  -p 8500:8500 \
  -p 8600:53/udp \
  progrium/consul -server -bootstrap -advertise ${HOSTIP} -log-level debug
```

Concul 自带界面，因此打开\${HOSTIP}:8500 就能看到其界面，Concul 启动后会开启两个端口，一个是 53/udp，还有一个是 8500/tcp，从界面上都能看到它们的状况。

接下来还需要启动 registrator 容器，目的是注册 Docker Container 的信息到 Concul 集群中。

```
$ docker run -d --restart=always \
  -v /var/run/docker.sock:/tmp/docker.sock \
  -h ${HOSTIP} \
  gliderlabs/registrator consul://${HOSTIP}:8500
```

启动一个 Nginx 服务器验证是否已经将自身信息注册到了 Concul 中，实现了自动发现的功能。

```
$ docker run -d -p 8080:80 --name nginx nginx:alpine
```

更复杂的应用场景，可以在 Concul 的官方文档中找到相关资料，本节不再展开。

18.4 第三方集群管理

18.4.1 Kubernetes 介绍

Kubernetes 是 Google 开发的一套开源的容器应用管理系统，用于管理应用的部署，维护和扩张，利用 Kubernetes 能方便地管理跨机器运行容器化的应用。

Kubernetes 也是用 Go 语言开发的，在 Github (<https://github.com/kubernetes/kubernetes>) 上可以找到源代码。

可以说在 Swarm 集成到 Docker 之前，Kubernetes 在集群管理方面优势明显。因为 Kubernetes 源于谷歌公司的内部容器管理系统 Borg，经过了多年生产环境的历练，功能非常强大。

Kubernetes 提供使用 Docker 对应用程序包装(package)、实例化(instantiate)、运行(run)，

并以集群的方式运行、管理跨机器的容器。与 Swarm 一样，Kubernetes 还可以配合服务发现工具解决 Docker 跨机器容器之间的通信问题。

Kubernetes 自带服务修复机制、应用高可用，这一点 Swarm 在集成到 Docker 之后也提供。Kubernetes 提出了不少的新概念，还支持安装 UI 的插件，如图 18.2 所示，来管理整个系统。

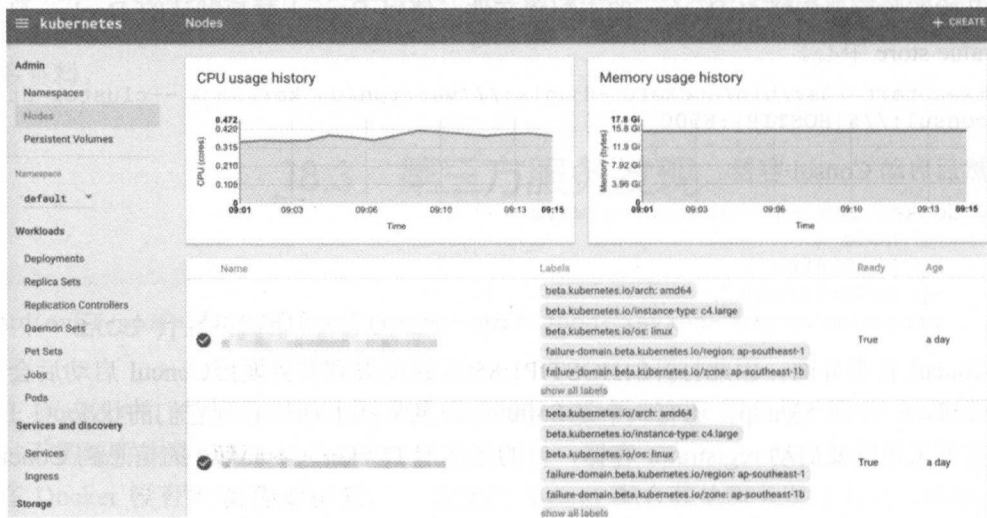


图 18.2 Kubernetes 管理界面

在 CentOS 中安装 Kubernetes。

```
$ yum install -y socat
```

添加源：

```
$ cat <<EOF> /etc/yum.repos.d/k8s.repo
[kubelet]
name=kubelet
baseurl=http://files.rm-rf.ca/rpms/kubelet/
enabled=1
gpgcheck=0
EOF
```

安装 Kubernetes。

```
$ yum makecache
$ yum install -y kubelet kubeadm kubectl kubernetes-cni
```

启动服务：

```
$ systemctl enable kubelet
$ systemctl start kubelet
```

然后使用 `kubeadm init` 命令创建集群。

```
$ kubeadm init --api-advertise-addresses=10.6.0.140
.....
Kubernetes master initialised successfully!
```

You can now join any number of machines by running the following on each node:

```
kubeadm join --token 8609e3.c2822cf312e597e1 10.6.0.140
```

查看集群状态:

```
$ kubectl get node
NAME           STATUS    AGE
k8s-master     Ready     1d
k8s-node-1     Ready     1d
k8s-node-2     Ready     1d
```

Kubernetes 功能之强大足够讲一本书, 本节只算一个介绍, 如果读者有兴趣可以在网络上找到大量资料, 毕竟这是目前容器集群管理方面最流行的工具之一。

18.4.2 Mesos Shipyard 介绍

如果说到集群管理三巨头的话, Kubernetes、Swarm 算两个名额, 而 Mesos 自然就是最后一个巨头了。

Mesos 是为软件定义数据中心而生的操作系统, 跨数据中心的资源在这个系统中被统一管理。Mesos 可以把不同机器的计算资源统一管理, 就像同一个操作系统, 用于运行分布式应用程序。Mesos 的初衷并非管理容器, 只是随着容器的发展, Mesos 加入了容器的功能。

Mesos 相比 Kubernetes 和 Swarm 更为成熟, 但是 Mesos 主要解决的是操作系统级别的抽象, 并非为了容器专门设计, 如果用户除了容器之外, 还要集成其他的应用, 例如 Hadoop、Spark、Kafka 等, Mesos 更为合适。

Mesos 是一个更重量级的集群管理平台, 功能更丰富, 当然很多功能要基于各种 Framework。Mesos 的扩展性非常好, 最大支持 50000 节点, 如果对扩展性要求非常高的话那么 Mesos 是最佳选择。

18.5 本章小结

本章主要讲述了 Swarm 集群网络的创建与部署, 介绍了 Swarm 的常规应用, 包括 Swarm 的服务发现、负载均衡等, 然后使用 Swarm 来配置跨主机容器网络, 并在上面部署应用, 在本章的最后还介绍了其他容器网络管理工具与集群管理工具。

第 19 章 Docker 安全

Docker 的安全问题一直比较突出，随着 Docker 的流行，Docker 安全方面的挑战也越来越大，我们不得不重视容器的安全问题。Docker 发展史中曾发生过一次比较严重的容器逃逸事件，影响 1.0 版本之前的 Docker，该项目可以逃出容器资源限制遍历宿主机文件系统，这给 Docker 商业化带来了不小阻力，也让人们认识到了 Docker 容器技术在安全方面的短板。

本章将围绕容器安全、镜像安全、仓库安全以及 Docker daemon 安全这 4 个方面来展开，介绍容器的资源限制等安全策略、镜像的校验传输加密、仓库的认证体系完善，以及 Docker daemon 的权限控制和 Socket、UNIX 传输安全。

19.1 Docker 安全机制

在了解如何解决问题之前，首先来了解 Docker 是如何保证安全的，以及在哪些方面还存在需要注意的问题。

因为网络安全前面已经讲过了，本章不再提及。网络方面如果使用 Swarm 创建的集群，可以为 Swarm 配置 TLS (<https://docs.docker.com/swarm/configure-tls/>)，如果使用第三方集群工具则需要读者自己找资料配置相关设置。而 Machine 的指令默认已经使用了自签名的 TLS 协议。

19.1.1 Daemon 安全

我们知道 Docker daemon 实际上是一个守护态的程序，用户操作都是通过 Docker client 或者 REST API 发送给 Docker daemon 的，在这个过程中，用户与 Docker daemon 的连接必须是可信的。例如，用户使用 `docker run -e PASSWORD ****` 这样的方式启动容器时，密码就暴露在 bash 等环境中，传输过程也有可能被截获。

Docker daemon 默认是使用 UNIX 域套接字的方式与用户通信，这个过程相对来说是安全的，因为只有进入 daemon 宿主机并且获得授权才可以操作 Docker。因此，相对的使用 TCP 方式与外界通信的过程就显得不是那么安全，因为可以访问到 Docker daemon 的机器都可以成为潜在的攻击者。攻击的方式可能是截获传输内容，也可能是通过 TCP 连接操作 Docker daemon 甚至获取宿主机 root 权限。

例如，使用 `docker daemon -H` 参数可以发布一个 Socket 地址，以供远程管理容器，但是也给黑客敞开了控制宿主机的门户，例如，创建一个容器挂载到 `ssh` 目录，从而实现免密码登录。

为了提高使用 TCP 通信方式的安全性，Docker 为用户提供了 TLS 加密传输的方法，在 Docker daemon 中有以下 4 个参数。

- `--tlsverify`: 安全传输检验。
- `--tlscacert=ca.pem`: 信任的证书。
- `--tlscert=server-cert.pem`: 服务证书。
- `--tlskey=server-key.pem`: 服务器或客户端密钥。

例如，使用下面的方式启动 Docker daemon 就是相对比较安全的。

```
$ docker daemon \
  --tlsverify \
  --tlscacert=ca.pem \
  --tlscert=server-cert.pem \
  --tlskey=server-key.pem \
  -H=0.0.0.0:2376
```

当然别忘了，在客户端中使用同样的方式发送命令（\$HOST 表示客户端所处网络中运行 daemon 的主机名）。

```
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --tlskey =
key.pem \
  -H=$HOST:2376 version
```

至于如何获得这些证书，前面私有仓库章节有介绍，如果读者想知道更详细的配置过程，可以访问官方文档，地址为 <https://docs.docker.com/engine/security/https/>。

此外，Docker daemon 以 root 权限运行，这意味着有一些问题需要格外小心。需要注意以下几点：

- 当 Docker 允许与非 root 用户创建的容器目录共享而没限制其访问权限时，Docker daemon 的控制权应该只给授权用户，也就是在安装完 Docker 时提示的那句话。仅授权特定用户免 sudo 操作执行 Docker 命令。
- REST API 支持 Unix sockets，从而防止了 cross-site-scripting 攻击。但是 REST API 的 HTTP 接口应该在可信网络或者 VPN 下使用。
- 在服务器上单独运行 Docker 时，需要与其他服务隔离。

19.1.2 容器与镜像安全

前面提到容器技术的原理是通过 cgroup 和 namespace 这两大特性来达到容器资源隔离的，而容器与宿主机是公用内核的，所以容器技术的攻击面比其他虚拟化技术要大。

此外，如果宿主机内核崩溃，那么在机器上的全部容器都会崩溃。而虚拟机一般不会因为宿主机内核问题而停止工作。

目前 namespace 在隔离上不算完善，比如 user namespace 还比较年轻，除此之外对于未隔离的内核资源（如 procfs 与 syslog 等信息）也会影响到容器安全。

关于 Docker 镜像的安全，在 Docker 1.8 之后提供了一个 Docker TUF（可信镜像及升级框架）功能，该功能使得用户可以校验镜像的发布者。

当镜像 push 到仓库时，Docker 会使用密钥对镜像进行签名，用户拉镜像到本地时，Docker 会使用公钥来校验镜像是否与发布者的一致。

在 docker run 中提供了容器内核功能配置接口，可以在创建容器时使用。在前面的章

节有简单介绍。借助 Linux 功能，可以分离 root 权限，形成更小的特权群。

目前，在默认情况下，Docker 容器只拥有以下功能。

```
CHOWN, DAC_OVERRIDE, FSETID, FOWNER, MKNOD, NET_RAW,
SETGID, SETUID, SETFCAP, SETPCAP, NET_BIND_SERVICE,
SYS_CHROOT, KILL, AUDIT_WRITE
```

19.2 Docker 资源控制

由于 Docker 中安全问题最突出的是容器方面，因此在 Docker 社区中讨论最多的安全问题就是容器安全，但也因此积累了最多的经验，本节将对业界的经验做些总结（虽然不全，但应该够用）。

19.2.1 限制 CPU

我们早就知道 cgroup 可以限制 CPU 与内存等资源的使用，但是没有实际应用过它来限制容器资源，本节就 CPU 限制做简单演示。

在 Docker 基础章节中曾解释过 docker run 的一些参数，其中就有 CPU 限制的参数。

--cpu-percent int	CPU percent (Windows only)
--cpu-period int	Limit CPU CFS (Completely Fair Scheduler)
period	
--cpu-quota int	Limit CPU CFS (Completely Fair Scheduler)
quota	
-c, --cpu-shares int	CPU shares (relative weight)
--cpuset-cpus string	CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems string	MEMs in which to allow execution (0-3, 0,1)

- **--cpu-percent**: 设置 CPU 使用率的百分比，只有 Windows 下才能起效。
- **--cpu-period**: 用来指定容器对 CPU 的使用要在多长时间内做一次重新分配。
- **--cpu-quota**: 用来指定在这个周期内，最多可以有多少时间用来运行这个容器。
- **--cpu-shares**: 用来设置容器对 CPU 使用的权重，默认情况下所有容器的 share（理解为权重）是相同的，也就是所有容器有相同的权重，在所有容器一起竞争资源时，最终得到的资源是相同的。这个 shares 是一个相对的值，如 A 和 B 两个容器，A 配置的是 1024，B 配置的是 512，那么 A 最大可以使用的 CPU 资源是 B 的两倍。还有一点要注意的是这种配置是有弹性的，如果 A 容器一直闲着，那 B 容器是可以使用空闲资源的。
- **--cpuset-cpus**: 可以绑定指定容器使用指定 CPU。
- **--cpuset-mems**: 只应用于 NUMA 架构的 CPU 生效。

实际上，所谓的 CPU 限制，不可能精确限制容器使用多少 GHz 的 CPU，而是相对地设置容器使用 CPU 的权重。也就是说，CPU 资源是无法预先精确分配好的，而是根据优先度获得 CPU 的资源。

例如，容器的默认权重是 1024，现在有两个容器在竞争 CPU 资源，默认情况下，两个容器使用 CPU 资源应该是平分的。

现在假设其中一个容器的 CPU 权重是 512，那么它相对于另一个容器而言，只能获得

对方一半的 CPU 资源。假设 CPU 资源总共有 100 份的话，那么限制的容器会获得 33.3 份，而不限容器可以获得 66.6 份。

上面只是假设两个容器竞争 CPU 资源的情况，事实上如果不限资源的容器没有使用 CPU 资源的话，那么限制使用 CPU 资源的容器也会获得全部的 CPU 资源。

设置容器 CPU 权重的命令是：

```
$ docker run --rm -it -c 100 ubuntu bash
```

上面设置容器的 CPU 权重为 100。

设定周期内容器对 CPU 的使用时间上限：

```
$ docker run --rm -it --cpu-quota 250000 --cpu-period 500000 ubuntu bash
```

上面表示该容器在 0.5 秒之内对 CPU 的使用时间上限是 0.25 秒。

19.2.2 限制内存

限制内存也是 Docker 容器运行时经常需要限制的一项硬件资源，使用 `-m` 可以限制容器使用内存的最高值，在 Linux 中还有一个 `swap`（虚拟内存），可以通过 `--memory-swap` 指定虚拟内存使用量的上限。

在仅使用 `-m` 指定内存使用上限而不指定 `--memory-swap` 来限制虚拟内存，可能会导致宿主机的虚拟内存被完全占用，因此最好同时指定两项的值。例如：

```
$ docker run --rm -it -m 400m ubuntu bash
```

上面限制了容器最大使用内存上限为 400 MB，虚拟内存最大使用上限为 800 MB。

如果 `--memory-swap` 设置小于 `--memory` 则设置不生效，使用默认设置，默认情况下，`--memory-swap` 会被设置成 `memory` 的两倍。

`--memory-swappiness=0` 表示禁用容器 `swap` 功能（这点不同于宿主机，宿主机 `swappiness` 设置为 0 也不保证 `swap` 不会被使用）。

```
$ docker run -it --rm -m 100M --memory-swappiness=0 ubuntu-stress:latest /bin/bash
```

`--memory-reservation ...` 选项可以理解为内存的软限制。如果不设置 `-m` 选项，那么容器使用内存可以理解为是不受限的。按照官方的说法，`memory reservation` 设置可以确保容器不会长时间占用大量内存。

19.2.3 限制 I/O

先来看 Docker 提供的容器 I/O 资源限制参数有哪些。

```
$ docker help run | grep -E 'bps|IO'
```

```
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

<code>--blkio-weight</code>	Block IO (relative weight), between 10 and 1000
<code>--blkio-weight-device=[]</code>	Block IO weight (relative device weight)
<code>--device-read-bps=[]</code>	Limit read rate (bytes per second) from a device
<code>--device-read-iops=[]</code>	Limit read rate (IO per second) from a device
<code>--device-write-bps=[]</code>	Limit write rate (bytes per second) to a device
<code>--device-write-iops=[]</code>	Limit write rate (IO per second) to a device

默认所有的容器对于 I/O 操作都拥有相同优先级。可以通过 `--blkio-weight` 修改容器权

重。`--blkio-weight` 权重值在 10~1000 之间。`--blkio-weight-device` 可以指定某个设备的权重大小。例如：

```
$ docker run -it --rm --blkio-weight 100 ubuntu bash
```

`--device-read-bps`、`--device-write-bps`、`--device-read-iops`、`--device-write-iops` 这 4 项用于限制容器的读写速度，单位可以是 KB、MB、GB 的正整数。例如：

```
$ docker run -it --rm --device-write-bps /dev/sda:1mb ubuntu bash
```

19.2.4 文件系统防护

在前面数据卷等章节中多次提到了文件系统保护的措施，Docker 可以设置容器的根文件系统为只读模式。只读模式下，即使容器与宿主机使用同一个文件系统，容器也没有写的权限，不会影响宿主机文件系统。设置只读模式，可以在 `docker run` 中添加 `--read-only` 参数。

此外，数据卷挂载中也可以使用 `-v /path:/path:ro` 的形式挂载为只读模式。

19.2.5 镜像瘦身神器 Docker Slim

Docker 的 Logo 是一只鲸鱼，但是用户的 Docker 容器可不需要鲸鱼这么大的体积。Docker-slim 是“容器的神奇减肥药”，它允许用户分析容器镜像并删减多余的东西。

容器中不需要的依赖和模块会增加容器的体积，使用 Docker Slim 可以把一个 Python 容器样本大小从大约 433MB 减少到 15.97MB，把一个 Java 应用样本大小从 743MB 变为 100.3MB。该分析会展示除去实际运行必需的软件包，有哪些是非必需的依赖和模块，用户可以使用这个信息来自己执行清理工作。

安装 Docker Slim，地址为 https://github.com/docker-slim/docker-slim/releases/download/1.17/dist_linux.zip。解压到设定的目录，然后使用 `chmod` 赋予执行权限。

使用 Slim 的格式如下：

```
$ ./docker-slim [info|build|profile] \
  [--http-probe|--remove-file-artifacts] \
  <IMAGE_ID_OR_NAME>
```

例如，检查一个 Node.js 镜像的结构。

```
$ ./docker-slim build --http-probe my/sample-node-app
```

更多示例，可以使用官方的示例文件来学习。

```
$ git clone https://github.com/docker-slim/docker-slim.git
```

19.2.6 强制访问控制工具 SELinux 或 AppArmor

SELinux 是由内核实现的 MAC（强制访问控制），通过访问控制的安全策略，可以配置 Linux 内核安全模块，常用的工具有 SELinux 和 AppArmor 等，从而实现强制性的访问控制（MAC），用以将进程约束在一套有限的系统资源或权限中。

如果已经安装并配置过 SELinux，那么可以在容器中使用 `setenforce 1` 来启用。Docker

守护进程的 SELinux 功能默认是禁用的，需要使用 `--selinux-enabled` 命令来启用。

容器的标签限制可使用 `--security-opt` 加载 SELinux 或者 AppArmor 的策略进行配置。

例如：

```
$ docker run --security-opt=secdriver:name:value -i -t centos bash
```

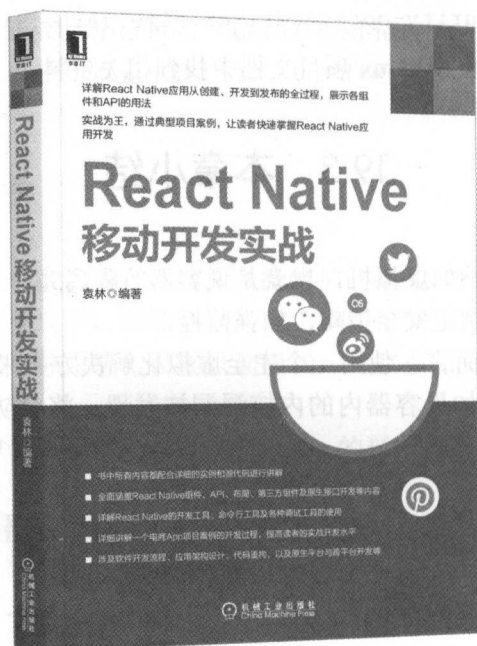
更详细的资料，可以在 SELinux 项目文档中找到相关资料。

19.3 本章小结

容器技术的隔离性远不如虚拟机，也就是说容器的隔离并非万能的，用户应该评估自己的需要，针对自身需求制定安全策略，加强监控。

对于一般的技术团队而言，使用一个完全虚拟化解方案来容纳 Docker，如 KVM，是一个不错的防御手段，如果容器内的内核漏洞被发现，将会防止其从容器扩大到宿主机上。

推荐阅读



React Native移动开发实战

作者：袁林 书号：978-7-111-57179-7 定价：69.00元

**详解React Native应用从创建、开发到发布的全过程，展示各组件和API的用法
实战为王，通过典型项目案例，让读者快速掌握React Native应用开发**

本书以实战开发为主旨，以React Native应用开发为主线，以iOS和Android双平台开发为副线，通过完整的电商类App项目案例，详细地介绍了React Native应用开发所涉及的知识，让读者全面、深入、透彻地理解React Native的主流开发方法，从而提升实战开发水平和项目开发能力。

本书共12章，分为4篇，涵盖的主要内容有搭建开发环境、Nuclide、各种命令行工具（Git、Node.js）、布局与调试、组件、API、第三方组件、基于Node.js的服务器、fetch API、AsyncStorage/SQLite/Realm数据库存储、原生平台接口开发、redux开发框架、应用打包与发布、热更新与CodePush等。

本书适合iOS和Android原生平台应用开发者，以及有兴趣加入移动平台开发的JavaScript开发者阅读。当然，本书也适合相关院校和社会培训学校作为移动开发的教材使用。

作者简介

黄靖钧

全栈开发者，热衷开源技术。长期以来一直使用容器技术作为应用部署方案，在Docker容器实战方面经验丰富。有多年的大规模集群管理经验。曾经从事PaaS与CaaS项目开发。现专注于Serverless与SDN等领域研究。

欢迎IT领域的各位技术牛人洽谈出书事宜。如果有写书或投稿意向，请加QQ或者微信具体商谈。

QQ: 627173439

微信: oyzx_sp

Docker从入门到实战

Introduction and Advanced Usage of Docker

本书核心内容

容器技术与Docker概念

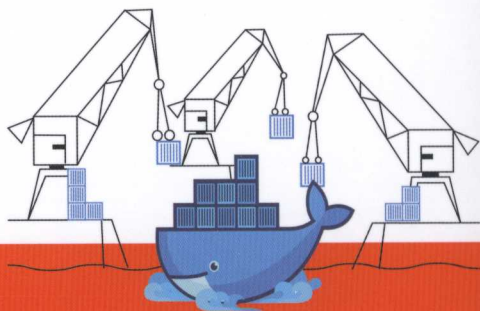
- 认识容器技术
- Docker基本概念
- 安装和测试Docker

Docker基础知识

- Docker操作命令
- 镜像的结构
- 镜像的拉取、修改与删除
- 镜像的体积控制
- 镜像的推送
- 容器的迁移与提交
- 镜像仓库
- 数据卷操作
- 容器网络操作

Docker进阶实战

- 操作系统镜像构建
- Web服务镜像构建
- 数据库镜像构建
- Compose与Machine
- 编程语言
- Docker API
- 私有仓库的部署和使用
- 集群网络
- 容器安全
- Docker网络生态



本书源代码获取方式

本书涉及的源代码文件需要读者下载。请在www.hzbook.com网站上搜索到本书页面，然后按照页面上的下载说明下载即可。



上架指导：计算机/网络管理

ISBN 978-7-111-57328-9



9 787111 573289 >

定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn